ΕΛΕΥΘΕΡΙΟΣ ΜΩΥΣΙΑΔΗΣ
Αναπληρωτής Καθηγητής, Τμήμα Πληροφορικής, ΔΙΠΑΕ

ΧΑΪΡΗ ΚΙΟΎΡΤ
Εντεταλμένος Ερευνητής, Ερευνητικό Κέντρο Αθηνά.

Εισαγωγή στην Java

Από το Διαδικαστικό στο Αντικειμενοστρεφές μοντέλο

Εισαγωγή στην Java

Συγγραφή

Ελευθέριος Μωυσιάδης

Chairi Kiourt

Κριτικός αναγνώστης (style: SintelestesTitle)

Όνομα 1 (Κριτικός αναγνώστης) (style: Sintelestes)

Συντελεστές έκδοσης (style: SintelestesTitle)

Γλωσσική Επιμέλεια: Όνομα (style: Sintelestes)

Γραφιστική Επιμέλεια: Όνομα (style: Sintelestes)

Τεχνική Επεξεργασία: Όνομα (style: Sintelestes)

ΚΑΛΛΙΠΟΣ

Εθνικό Μετσόβιο Πολυτεχνείο

Ηρώων Πολυτεχνείου 9, 15780 Ζωγράφου

www.kallipos.gr

ISBN:

Βιβλιογραφική Αναφορά (για βιβλίο με 2 συγγραφείς): Μωυσιάδης, Ε., & Κιούρτ, Χ. (2022). *Εισαγωγή στην Java* [Νέα συγγράμματα για Προπτυχιακά μαθήματα]. Κάλλιπος, Ανοικτές Ακαδημαϊκές Εκδόσεις. http://hdl.handle.net/(συμπληρώνεται από την ΚΟΥ)

Στη γυναίκα μου, Ανθή
και στα παιδιά μου,
Αναστάση και Ειρήνη
Ε.Μ.

Στην κόρη μου, Ναρίν
και στη σύζυγό μου,
Σεβτζάν
Χ.Κ.

Πίνακας Περιεχομένων

Πίνακας συντομεύσεων-ακρωνυμίων

Ακρωνύμιο	Περιγραφή
JSE	Java Standard Edition
JEE	Java Enterprise Edition
JME	Java Micro Edition
JVM	Java Virtual Machine
JIT	Just In Time
JRE	Java Runtime Environment
JDK	Java Development Kit
IDE	Integrated Development Environments
API	Application Programming Interface
HTML	HyperText Markup Language
RAM	Random Access Memory
JDB	Java Data Base
RIA	Rich Internet Applications

XML	Extended Markup Language
JAF	JavaBeans Activation Framework
JPEG	Joint Photographic Experts Group
OCR	Optical Character Recognition

Πρόλογος

Κατά τη βιομηχανική επανάσταση, η ανθρωπότητα γνώρισε ραγδαία ανάπτυξη και πρόοδο. Εκείνη την περίοδο, ο άνθρωπος κατασκεύασε μηχανές που ήταν σε θέση να τον αντικαταστήσουν σε πλήθος χειρωνακτικών εργασιών με δραματικά αποτελέσματα στην αύξηση της παραγωγής. Στη σύγχρονη εποχή όμως, οι ρυθμοί ανάπτυξης ξεπερνούν κατά πολύ αυτούς της περιόδου εκβιομηχάνισης. Τώρα ο άνθρωπος κατασκευάζει μηχανές ικανές να υπολογίζουν, να βλέπουν, να λαμβάνουν αποφάσεις, ακόμη και να σκέπτονται ή να μιλούν.

Σήμερα, περισσότερο από κάθε άλλη εποχή της ανθρώπινης Ιστορίας, η ισχύς των νέων τεχνολογιών, θέτει μια σειρά από ερωτήματα, γεννάει ελπίδες και εγκυμονεί κινδύνους. Θα καταφέρουμε άραγε να αξιοποιήσουμε τις νέες τεχνολογίες αποκλειστικά για το καλό της ανθρωπότητας; Μήπως θα φτάσουμε σε αυτόν τον βαθμό τεχνητής ευφυΐας που η υποδούλωσή μας στις μηχανές θα καταστεί αναπόφευκτη; Μήπως οι ισχυροί του πλανήτη αξιοποιώντας την τεράστια ισχύ της τεχνολογίας καταφέρουν να ελέγξουν τους λαούς;

Τίποτα άλλο δεν μπορεί να εγγυηθεί την έκβαση της επανάστασης της Πληροφορικής παρά η πλατιά συμμετοχή όσων το δυνατόν περισσότερων πολιτών στη νέα γνώση. Όσο περισσότεροι είμαστε κοινωνοί των μυστικών της σύγχρονης τεχνολογίας τόσο ισχυρότερη είναι η άμυνά μας και τόσο μεγαλύτερη η ασφάλειά μας απέναντι σε τυχόν κακόβουλη αξιοποίησή της.

Αυτή η σκέψη αποτέλεσε το κύριο κίνητρο συγγραφής αυτού του βιβλίου. Πρόκειται για μια προσπάθεια διάχυσης της γνώσης στον νευραλγικό τομέα της Πληροφορικής, τον Προγραμματισμό. Ευελπιστούμε πως έτσι πραγματοποιούμε μια ελάχιστη συμβολή στην πρόοδο.

Αν οι ελπίδες μας έχουν κάποια βάση ή όχι, ο μόνος κατάλληλος για να το πει είστε εσείς, οι αναγνώστες αυτού του βιβλίου. Ανεξάρτητα από την ετυμηγορία σας, θέλουμε να σας ευχαριστήσουμε, γιατί χωρίς εσάς δεν θα είχε κανένα απολύτως νόημα η ύπαρξή του.

Εισαγωγή

Είναι κοινή αντίληψη πως η ανάπτυξη της πληροφορικής ακολουθεί πρωτόγνωρα ραγδαίους ρυθμούς. Μέχρι χθες δεν υπήρχαν κινητά τηλέφωνα, ενώ σήμερα είναι ολοκληρωμένοι υπολογιστές που αναγνωρίζουν το πρόσωπο του ιδιοκτήτη τους και του ανοίγουν μια πύλη εισόδου στο παγκόσμιο δίκτυο. Πέραν της ανάπτυξης της ίδιας της πληροφορικής, πληθώρα διεπιστημονικών πεδίων κάνουν την εμφάνισή τους. Η Βιοπληροφορική, η Μηχατρονική, οι Τηλεπικοινωνίες, οι Τεχνολογίες Πληροφορικής και Επικοινωνιών στην Εκπαίδευση, η Τεχνογλωσσία που συνδυάζει Γλωσσολογία και Πληροφορική, η Νανοτεχνολογία, η Νομική Πληροφορική, η Ρομποτική και άλλοι κλάδοι στους οποίους η Πληροφορική διαδραματίζει σημαίνοντα ρόλο.

Ακόμη και επιστήμες που εκ πρώτης όψεως δεν σχετίζονται με την Πληροφορική, έχουν ωφεληθεί από τις εφαρμογές της. Στους ερευνητές όλων των ειδικοτήτων, προσφέρει άμεση πρόσβαση στο σύνολο σχεδόν της ανθρώπινης γνώσης. Στην Ιατρική, προγραμματιζόμενα ρομπότ συμμετέχουν σε κρίσιμες επεμβάσεις και έμπειρα συστήματα βοηθούν σε έγκυρες διαγνώσεις. Στη Νομική, βάσεις δεδομένων εξυπηρετούν την πρόσβαση σε μεγάλες βάσεις πληροφοριών. Το ηλεκτρονικό εμπόριο επιταχύνει τις συναλλαγές αυξάνοντας το παραγόμενο προϊόν. Η Ωκεανογραφία εξοπλίζεται με νέα συστήματα που διευκολύνουν τη μελέτη των ωκεανών και των βυθών.

Ήδη με την ανάπτυξη της Τεχνητής Νοημοσύνης, τα πρώτα συστήματα που έχουν την ικανότητα να μαθαίνουν έχουν τεθεί σε λειτουργία. Ήδη ο άνθρωπος είναι κατώτερος σκακιστής σε σχέση με τον Υπολογιστή. Φαίνεται πως η Ανολοκλήρωτη Επανάσταση [1] γίνεται πραγματικότητα. Οι σύγχρονοι Υπολογιστές αναγνωρίζουν τον ιδιοκτήτη τους από το πρόσωπό του, αναγνωρίζουν τα δακτυλικά του αποτυπώματα, οι χρήστες μπορούν και επικοινωνούν με τους υπολογιστές στη δική τους ανθρώπινη γλώσσα.

Το μέλλον διαγράφεται ακόμη πιο δυναμικό. Σύμφωνα με μια δημοσίευση [2] της Kaspersky, μεγάλης εταιρείας Πληροφορικής, σε τρεις δεκαετίες, όλες οι βαριές δουλειές θα γίνονται από ρομπότ, τα σπίτια πλήρως αυτοματοποιημένα θα διεκπεραιώνουν εργασίες από την αναγκαία μαγειρική μέχρι τη διαχείριση των προμηθειών, τα αυτοκίνητα θα κινούνται αυτόνομα, οι κατασκευές επίσης θα αυτοματοποιηθούν με τη βοήθεια 3D εκτυπωτών. Ακόμη και η ίδια η ανθρώπινη φύση θα μεταβληθεί, καθώς βαδίζουμε προς τον μηχανικό άνθρωπο, δηλαδή τον άνθρωπο που θα διαθέτει τεχνητά μέλη και όργανα.

Στη βάση όλων αυτών των κατακλυσμιαίων αλλαγών βρίσκεται ο προγραμματισμός. Ο προγραμματισμός αποτελεί τη βασική οδό επικοινωνίας του ανθρώπου με τον υπολογιστή, τον τρόπο με τον οποίο ο άνθρωπος διαμορφώνει τον "νου" του άψυχου υλικού.

Η ανάπτυξη και η εξέλιξη του προγραμματισμού έχει επίσης γνωρίσει δραματικά ταχείς ρυθμούς. Πριν από μόλις τριάντα χρόνια, μία γλώσσα προγραμματισμού συμπυκνωνόταν σε ένα βιβλίο περιορισμένου όγκου και ένας επαγγελματίας του χώρου μπορούσε να καυχηθεί ότι γνωρίζει όλες τις λεπτομέρειες της γλώσσας. Σήμερα, οι περισσότερες γλώσσες προγραμματισμού έχουν μετατραπεί σε πλαίσια ανάπτυξης. Τα πλαίσια αυτά, πέραν του πυρήνα μιας γλώσσας, ενσωματώνουν βιβλιοθήκες ικανές να υποστηρίξουν πάρα πολλές λειτουργίες, από διαχείριση βάσεων δεδομένων μέχρι προγραμματισμό ενσωματωμένων συστημάτων, από τη διεθνοποίηση των προγραμμάτων μέχρι τις κανονικές εκφράσεις, από υπηρεσίες εξουσιοδότησης και αναγνώρισης μέχρι την προσαρμοσμένη δικτύωση. Έτσι οι λεπτομέρειες ενός πλαισίου, όπως της Java για παράδειγμα, είναι αδύνατο, λόγω όγκου, να συμπεριληφθούν σε ένα βιβλίο και ο καλύτερος επαγγελματίας δεν γνωρίζει με λεπτομέρεια παρά μόνο ένα ποσοστό από τις διαθέσιμες δυνατότητες.

Ο τεράστιος αυτός όγκος συνιστά μια πρόκληση και στον εκπαιδευτικό τομέα. Σε ποια επιμέρους αντικείμενα μπορεί να διαχωριστεί η αναγκαία ύλη; Με ποια σειρά θα πρέπει να δοθούν τα επιμέρους αντικείμενα στον εκπαιδευόμενο; Πώς θα πρέπει να διαμορφωθεί η ύλη σε κάθε επιμέρους αντικείμενο; Αυτά είναι κάποια από τα θεμελιώδη ερωτήματα στα οποία οφείλει να απαντήσει κάθε πρόγραμμα σπουδών εφόσον περιλαμβάνει προγραμματισμό. Για να δοθεί μια εικόνα στον αναγνώστη αναφέρουμε ως παράδειγμα ότι στο Τμήμα Πληροφορικής του ΔΙΠΑΕ υπάρχουν δέκα (10) μαθήματα εκμάθησης αποκλειστικά του Προγραμματισμού. Πιο συγκεκριμένα, Εισαγωγή στον προγραμματισμό με C++, Εισαγωγή στην Java, Αντικειμενοστρεφής Προγραμματισμός, Προγραμματισμός Διεπαφής Χρήστη, Προηγμένα Θέματα Προγραμματισμού, Προγραμματισμός του Παγκόσμιου Ιστού, Ανάπτυξη Προηγμένων Εφαρμογών Κινητών Συσκευών, Λογική και Λογικός Προγραμματισμός, Προγραμματισμός Δικτύων και τέλος Σχεδιαστικά Πρότυπα. Πέραν αυτών ένα μεγάλο πλήθος μαθημάτων, όπως η Τεχνητή Νοημοσύνη, οι Βάσεις Δεδομένων, η Τεχνητή Όραση, τα Αυτοκινούμενα Ρομπότ, τα Νοήμονα Ρομπότ, τα Λειτουργικά Συστήματα, οι Αλγόριθμοι Βελτιστοποίησης, οι Αλγόριθμοι Βιοπληροφορικής κ.ά συμπεριλαμβάνουν τον προγραμματισμό,

όχι απλώς ως εφαρμογή της διδακτέας ύλης των μαθημάτων που αφορούν αποκλειστικά τον Προγραμματισμό, αλλά διδάσκοντας νέα προγραμματιστικά εργαλεία απαραίτητα για κάθε ένα από αυτά τα αντικείμενα.

Μέσα σε αυτήν την πληθώρα θεμάτων, το εγχειρίδιο αυτό έχει ως στόχο να καλύψει το Διαδικαστικό και το Αντικειμενοστρεφές μοντέλο με την Java. Σε αυτό το επίπεδο, κρίσιμο είναι το ερώτημα της διάρθρωσης της ύλης. Η αλήθεια είναι ότι κυκλοφορούν αρκετά βιβλία που καλύπτουν τη συγκεκριμένη θεματική περιοχή. Στη μεγάλη πλειοψηφία τους παρουσιάζουν την ύλη παράλληλα, εμπλέκοντας έννοιες του διαδικαστικού με του αντικειμενοστρεφούς μοντέλου. Δεν μπορεί κανείς να θεωρήσει ότι πρόκειται για λανθασμένη επιλογή. Πολλοί συγγραφείς θεωρούν ότι κάποιες έννοιες πρέπει να διδαχθούν από την αρχή, καθώς έτσι διευκολύνεται η έγκαιρη εμπέδωσή τους. Ωστόσο, η δική μας προσέγγιση διαφέρει. Ιστορικά, πρώτα αναπτύχθηκε το διαδικαστικό μοντέλο και στη συνέχεια το αντικειμενοστρεφές. Αυτό δεν σημαίνει όμως πως το διαδικαστικό μοντέλο είναι παρωχημένο. Αντίθετα, εμπεριέχεται και αποτελεί τη βάση και του αντικειμενοστρεφούς αλλά και σχεδόν όλων των αντικειμένων προγραμματισμού που αναφέρθηκαν προηγουμένως. Επιπλέον, το διαδικαστικό μοντέλο είναι αυτοδύναμο, μπορεί να διδαχθεί αυτόνομα και σε γενικές γραμμές, οι έννοιές του είναι απλούστερες και άρα πιο προσιτές στον εκπαιδευόμενο προγραμματιστή. Βέβαια, η Java είναι μια καθαρά αντικειμενοστρεφής γλώσσα. Αυτό όμως δεν σημαίνει πως δεν συμπεριλαμβάνει το διαδικαστικό μοντέλο. Αξιολογώντας αυτά τα δεδομένα, αποφασίσαμε να οργανώσουμε την ύλη κατά τέτοιο τρόπο ώστε να εξαντλήσουμε πρώτα τη μελέτη όλων των διαδικαστικών δομών και εννοιών και στη συνέχεια να προχωρήσουμε στο αντικειμενοστρεφές μοντέλο. Ευελπιστούμε πως αυτή η διάρθρωση διευκολύνει την πρόσληψη της ύλης και την κατανόηση των εννοιών. Σε αυτό το πλαίσιο προσπαθήσαμε να δώσουμε στον αναγνώστη όλες τις αναγκαίες πληροφορίες με τη σωστή σειρά, αποφεύγοντας τη φλυαρία που κατά την άποψή μας κάνει τη μελέτη κουραστική.

Πέρα από αυτήν τη βασική επιλογή, θα θέλαμε να επιστήσουμε την προσοχή του αναγνώστη στις ακόλουθες κατευθυντήριες γραμμές μελέτης:

Ορολογία. Δίνουμε μεγάλη σημασία στη γνώση της ορολογίας του προγραμματισμού. Με λίγο φιλοσοφική διάθεση μπορούμε να πούμε ότι γλώσσα και σκέψη δεν είναι παρά οι δύο όψεις του ίδιου νομίσματος. Άραγε μπορούμε να επεξεργαστούμε σύνθετες σκέψεις χωρίς να τις οργανώσουμε λεκτικά; Μπορούμε να σκεφτούμε κάτι έστω και απλό χωρίς να το αναπαραστήσουμε νοητικά με λέξεις; Πώς θα κατανοήσουμε μια έννοια στον Προγραμματισμό που περιέχει στον ορισμό της άλλες απλούστερες έννοιες αν δεν τις έχουμε ήδη κάνει κτήμα μας;

Πράξη. Η πράξη είναι εξαιρετικά σημαντική στην εκμάθηση του Προγραμματισμού. Όσα βιβλία και να διαβάσω, όση αποφασιστικότητα και να διαθέτω για να εντρυφήσω στις έννοιες του προγραμματισμού, ελάχιστα θα καταφέρω αν δεν προγραμματίζω. Προγραμματίζοντας έχω την ευκαιρία να βλέπω τα λάθη και τις παραλείψεις μου, να αναπτύσσω την απαραίτητη αλγοριθμική σκέψη και να εξοικειώνομαι με τις αναγκαίες έννοιες στην πράξη. Όμως η πράξη δεν θα πρέπει να είναι τυχαία και άναρχη, αντίθετα θα πρέπει να διακρίνεται από συστηματικό χαρακτήρα. Θα δώσω ένα παράδειγμα για να εξηγήσω σαφώς τι εννοώ. Συχνά βρίσκομαι μάρτυρας του εξής περιστατικού: Το πρόγραμμα κάποιου εκπαιδευόμενου παρουσιάζει κάποιο πρόβλημα. Ο εκπαιδευόμενος αρχίζει να δοκιμάζει τυχαίες αλλαγές στον κώδικα με σκοπό να εξαλείψει το πρόβλημα. Κάποια στιγμή, το πρόβλημα δεν εμφανίζεται πλέον και ο εκπαιδευόμενος θεωρεί ότι το πρόβλημα λύθηκε. Πρόκειται για λανθασμένη εφαρμογή της μεθόδου ‘προσπάθησε και κάνε λάθος’ (try and error). Με αυτόν τον τρόπο είναι πιθανό πως το πρόβλημα δεν λύθηκε καν αλλά απλώς δεν εμφανίζεται κάτω από συγκεκριμένες προϋποθέσεις. Αν αυτές αλλάξουν, το πρόβλημα ενδέχεται να εμφανιστεί εκ νέου. Ακόμη όμως και αν το πρόβλημα λυθεί, ο εκπαιδευόμενος δεν κατανόησε τι ακριβώς δημιούργησε το πρόβλημα. Σε πρώτη ευκαιρία θα το επαναλάβει σε κάποιο άλλο σημείο του κώδικα. Αντί αυτής της πρακτικής, μελετήστε με προσοχή τα μηνύματα που αναφέρονται στο πρόβλημα και ψάξτε συστηματικά στον κώδικά σας μέχρι να καταλάβετε ποιο ακριβώς είναι το θέμα. Μόνο τότε προχωρήστε σε επέμβαση στον κώδικα, όταν είστε σίγουροι πως έχετε καταλάβει σωστά. Με αυτόν τον τρόπο, και θα παρέχετε πιο αξιόπιστη λύση, και θα ενσωματώσετε μια γνώση που θα αποδειχτεί σίγουρα χρήσιμη.

Θεωρία. Η πράξη είναι απαραίτητη. Το ίδιο απαραίτητη είναι και η θεωρία. Πολλοί αρχάριοι προγραμματιστές δεν δίνουν την πρέπουσα σημασία στη θεωρία. Η αλήθεια είναι πως στον προγραμματισμό, γνωρίζοντας μερικά βασικά στοιχεία μπορεί να επιχειρήσει κανείς να κάνει αρκετά πράγματα. Αυτό το χαρακτηριστικό, συχνά δίνει την εντύπωση πως δεν απαιτούνται και ιδιαίτερες γνώσεις για να προγραμματίσουμε. Πρόκειται όμως για εντελώς λανθασμένη αντίληψη. Δεν αρκεί να μπορώ να κάνω κάτι αν δεν γνωρίζω έναν σχετικά οικονομικό τρόπο για να το πετύχω. Δεν χρησιμεύει σε τίποτα να μπορώ να πάω από την Αθήνα στη Θεσσαλονίκη αν ο μόνος δρόμος που γνωρίζω περνάει από τον Βόρειο Πόλο. Η βαθιά

γνώση της θεωρίας είναι αυτή που θα διαφοροποιήσει τον αυριανό επαγγελματία προγραμματιστή από τον αιώνιο ερασιτέχνη.

Αγγλικά. Ένα άλλο σημείο στο οποίο θα ήθελα να επιστήσω την προσοχή στον επίδοξο εκπαιδευόμενο προγραμματιστή είναι η αναγκαιότητα γνώσης των Αγγλικών. Τυπικά, οι νέες εξελίξεις στον χώρο ανακοινώνονται στα Αγγλικά. Στη συντριπτική πλειοψηφία τους χώροι του διαδικτύου στους οποίους μπορείτε να λύσετε τις απορίες σας και να ενημερωθείτε σχετικά για ποικίλα θέματα Προγραμματισμού, χρησιμοποιούν τα Αγγλικά. Γνωρίζω ότι πολλοί σπουδαστές αντιμετωπίζουν δυσκολίες με τα Αγγλικά. Δυστυχώς όμως, στον τομέα του Προγραμματισμού, η γνώση των Αγγλικών είναι αναγκαία. Όμως αυτό δεν θα πρέπει να αποθαρρύνει κανέναν. Δεν απαιτείται να γνωρίζουμε Αγγλικά στο επίπεδο του να διαβάζουμε ή να γράφουμε περίπλοκα λογοτεχνικά ή φιλοσοφικά δοκίμια. Η ορολογία του χώρου είναι σχετικά περιορισμένη. Αν επιμείνετε να διαβάζετε προγραμματισμό στα Αγγλικά, σύντομα θα νοιώθετε άνετα και θα κατανοείτε τις σχετικές έννοιες. Εξάλλου, σήμερα, χάρη στην ανάπτυξη της Πληροφορικής, έχετε στη διάθεσή σας διάφορα γλωσσικά εργαλεία που μπορούν να σας βοηθήσουν σε αυτήν την προσπάθεια, π.χ. ο μεταφραστής της Google (Google Translator).

Στη συνέχεια, αυτό το εγχειρίδιο οργανώνεται ως ακολούθως: Τα κεφάλαια 1 έως και 9 μας εισάγουν στον διαδικαστικό προγραμματισμό με Java. Τα υπόλοιπα κεφάλαια αφορούν το αντικειμενοστρεφές μοντέλο. Αμέσως μετά το κεφάλαιο 9, παρατίθεται μια σειρά από κριτήρια αξιολόγησης. Παρόμοια κριτήρια αξιολόγησης δίνονται και στο τέλος του βιβλίου. Τα κριτήρια αξιολόγησης έχουν αναπτυχθεί ώστε να στοχεύουν αφενός να πληροφορήσουν τον αναγνώστη κατά πόσο έχει κάνει κτήμα του την αντίστοιχη ύλη, αφετέρου να του επισημάνουν πως στον Προγραμματισμό υπάρχουν λεπτομέρειες που είναι αναγκαίο να γνωρίζουμε.

Με αυτές τις σκέψεις, κλείνω την εισαγωγή και σας καλωσορίζω στον δημιουργικό κόσμο του Προγραμματισμού!

Κεφάλαιο 1

Σύνοψη
Παρουσιάζονται καταρχάς ιστορικά στοιχεία σχετικά με τη δημιουργία, την εξέλιξη, τους στόχους και τις βασικές σχεδιαστικές επιλογές της Java. Στη συνέχεια, εξηγούνται οι βασικές έννοιες: Γλώσσα μηχανής (Machine language), Assembly, Μεταγλωττιστής (Compiler), Διερμηνευτής (Interpreter), Διασυνδέτης (Linker). Παρουσιάζεται η αρχιτεκτονική της Java και εξηγείται η λειτουργία των συστατικών της, της Εικονικής Μηχανής (Java Virtual Machine), του Περιβάλλοντος Εκτέλεσης (Java Runtime Environment), της Εργαλειοθήκης Ανάπτυξης (Java Development Kit), του μεταγλωττιστή Just In Time. Επισημαίνεται ο διαπλατφορμικός χαρακτήρας της Java. Επιπλέον, παρουσιάζονται οι βασικές αρχές σχεδίασης και οι τεχνολογίες που υποστηρίζονται στο βασικό πλαίσιο της Java. Πιο συγκεκριμένα, αναφέρονται το πλαίσιο σύνδεσης με βάσεις δεδομένων, η ενσωματωμένη βάση δεδομένων, το πλαίσιο ανάπτυξης για ενσωματωμένα συστήματα (Java SE Embedded), τα πλαίσια ασφάλειας (Security), τα πλαίσια διεθνοποίησης (Internationalization), ανάπτυξης εφαρμογών δικτύου (Custom Networking), διανομής (Deployment), κανονικών εκφράσεων (Regular Expressions), τα πλαίσια διαδικτυακών εφαρμογών (Java Servlet API, JavaServer Pages Technology, JavaServer Pages Standard Tag Library, JavaServer Faces Technology, Java Message Service API, JavaMail API and the JavaBeans Activation Framework), το πλαίσιο διαχείρισης XML αρχείων (Java API for XML Processing), τα πλαίσια ανάπτυξης εφαρμογών με παραθυρική διεπαφή (Swing και JavaFx) και εμπλουτισμένων εφαρμογών διαδικτύου (Rich Internet Applications), η δυνατότητα για ανάπτυξη εφαρμογών κινητών και διαπλατφορμικών εφαρμογών.

Προαπαιτούμενη γνώση
Δεν απαιτούνται ειδικές γνώσεις για αυτήν την ενότητα.

Λέξεις κλειδιά
Γλώσσα μηχανής, Μεταγλωττιστής, Εικονική Μηχανή

1 Ιστορικά στοιχεία και βασικές έννοιες

Δύο είναι τα βασικά συστατικά μέρη ενός υπολογιστικού συστήματος, το υλικό (hardware) και το λογισμικό (software). Το υλικό, όπως φανερώνει και το όνομά του περιλαμβάνει εκείνο το τμήμα που έχει υλική υπόσταση. Με άλλα λόγια, το υλικό είναι μια μηχανή. Ωστόσο, πρόκειται για μια ιδιαίτερη μηχανή, θεμελιωδώς διαφορετική από όλες τις μη υπολογιστικές μηχανές. Όλες οι άλλες μηχανές σχεδιάζονται για να εκτελούν μια προκαθορισμένη εργασία. Αντίθετα, το υλικό του υπολογιστή από μόνο του δεν εκτελεί καμία εργασία. Μπορεί όμως να εκτελέσει οποιαδήποτε εργασία του υποδειχθεί από το λογισμικό. Τι είναι όμως το λογισμικό; Ένας απλός ορισμός λέει πως το λογισμικό ενός υπολογιστή είναι το σύνολο των προγραμμάτων που τρέχουν σε αυτόν. Τι είναι όμως ένα πρόγραμμα; Ένα πρόγραμμα στην απλούστερη μορφή του, δηλαδή εκφρασμένο στη γλώσσα της μηχανής, είναι ένα σύνολο εντολών τέτοιων ώστε κάθε εντολή να είναι εκτελέσιμη από το υλικό. Πρόκειται για εντολές που αντιστοιχούν σε στοιχειώδεις λειτουργίες. Ωστόσο, οι εντολές αυτές μπορούν να οργανωθούν σε αναρίθμητους συνδυασμούς έτσι ώστε κάθε συνδυασμός να παράγει ένα εντελώς διαφορετικό αποτέλεσμα. Τελικά, η μηχανή καταφέρνει και εκτελεί πολλές και διαφορετικές μεταξύ τους εργασίες.

Έτσι λοιπόν με τη βοήθεια του λογισμικού, το υλικό αποκτά χαρακτηριστικά που αντίστοιχά τους συναντά κανείς μόνο στην ανθρώπινη σκέψη, όπως για παράδειγμα η λήψη αποφάσεων ανάλογα με τα διαθέσιμα δεδομένα. Από αυτό το σημείο ξεκινάει η μεγάλη περιπέτεια της σύγχρονης πληροφορικής η οποία από τη γέννησή της μέχρι σήμερα έχει γνωρίσει μια άνευ προηγουμένου ανάπτυξη, ενώ ταυτόχρονα συμβάλλει τα μέγιστα στην ταχύτατη εξέλιξη και των υπόλοιπων επιστημών.

Όπως όμως σε όλα τα μεγάλα επιτεύγματα της ανθρώπινης επινόησης, έτσι και στον τομέα της ανάπτυξης λογισμικού δεν έλειψαν τα προβλήματα. Σύντομα, μετά τη συγγραφή των πρώτων προγραμμάτων, έγινε αντιληπτή η ανάγκη να αναπτυχθούν γλώσσες με τις οποίες να μπορούν οι άνθρωποι να προγραμματίζουν εύκολα τους υπολογιστές. Μια από αυτές τις γλώσσες είναι και η Java.

Στη συνέχεια αυτού του κεφαλαίου, αφού δώσουμε συνοπτικά τα κύρια ιστορικά στοιχεία για τη γέννηση και εξέλιξη της Java, προχωρούμε στην παρουσίαση βασικών εννοιών αναγκαίων για την κατανόηση του προγραμματισμού.

1.1 Ιστορικά στοιχεία

Το 1991, ο James Gosling στην Sun Microsystems, ανέλαβε τον σχεδιασμό μιας νέας γλώσσας, προδρόμου της Java, με το όνομα Oak [3]. Στόχος του ήταν να αξιοποιηθεί η Oak για τον προγραμματισμό οικιακών συσκευών. Προέκυψε λοιπόν η ανάγκη ένα λογισμικό που θα γραφόταν για συγκεκριμένη συσκευή, π.χ. τηλεόραση ενός κατασκευαστή, να τρέχει και σε οποιοδήποτε άλλο μοντέλο του ίδιου ή άλλου κατασκευαστή. Από την ανάγκη αυτή πηγάζει ο διαπλατφορμικός χαρακτήρας της Java. Ήδη το 1994, η Oak είχε μετονομαστεί σε Java και άρχισε να διαμορφώνεται κατάλληλα ώστε να υποστηρίζει διαδικτυακές εφαρμογές. Σύντομα, οι κατασκευαστές λογισμικού πλοήγησης στο διαδίκτυο με πρώτη την Netscape Communications άρχισαν να υποστηρίζουν τη νέα γλώσσα [4].

Το 1996, η Sun Microsystems παρουσίασε την πρώτη δημόσια έκδοση της γλώσσας. Στο τέλος του 1998, η ίδια εταιρεία, παρουσίασε την Java 2 που περιλάμβανε τρεις διαφορετικές διαμορφώσεις, τη βασική, τη διαμόρφωση για εταιρείες και τη διαμόρφωση για προγραμματισμό συσκευών συμπεριλαμβανομένων των κινητών τηλεφώνων. Οι εκδόσεις αυτές εξελίχθηκαν και ονομάστηκαν το 2006 από την Sun ως Java Standard Edition (JSE), Java Enterprise Edition (JEE) και Java Micro Edition (JME), αντίστοιχα [5]. Το 2007 η Java υιοθετείται από την Google ως βασική γλώσσα για ανάπτυξη εφαρμογών Android [6]. Την ίδια εποχή περίπου, ο Chris Oliver στην εταιρεία SeeBeyond υλοποιεί ένα πλαίσιο ανάπτυξης παραθυρικών διεπαφών που ονόμασε F3. Το 2008, η Sun αγοράζει την SeeBeyond και προσθέτει στην οικογένεια της Java την εξέλιξη της F3 που ονόμασε JavaFx [7].

Από τη γέννησή της έως σήμερα η Java βελτιώνεται και επαυξάνεται διαρκώς. Η JSE κατά τον χρόνο συγγραφής αυτού του κειμένου βρίσκεται στην έκδοση 17. Οι κώδικες που παρουσιάζονται σε αυτό το βιβλίο μεταγλωττίζονται με επιτυχία από την έκδοση 8 ή μεταγενέστερη.

1.2 Βασικές Έννοιες

Γλώσσα Μηχανής: Η θεμελιώδης γλώσσα προγραμματισμού είναι η λεγόμενη Γλώσσα Μηχανής (Machine Language). Το "αλφάβητο" αυτής της γλώσσας είναι οι χαρακτήρες 0 και 1, δηλαδή τα δυαδικά ψηφία (binary digits ή bits). Με συνδυασμό των bits σχηματίζεται το λεξιλόγιο της μηχανής, γνωστό ως σύνολο εντολών (Instruction Set). Επομένως, τα προγράμματα που είναι γραμμένα σε γλώσσα μηχανής είναι ακολουθίες από bits,γεγονός που καθιστά τη συγγραφή αλλά και την ανάγνωσή τους ιδιαίτερα δύσκολη. Επιπλέον, το σύνολο των εντολών διαφέρει ανάλογα με την αρχιτεκτονική της μηχανής. Αυτό σημαίνει ότι η ίδια λειτουργικότητα είναι αναγκαίο να υλοποιηθεί ξεχωριστά για κάθε διαφορετική μηχανή. Τέλος, τα προγράμματα σε γλώσσα μηχανής είναι απευθείας εκτελέσιμα (executable) από τον υπολογιστή.

Assembly: Προκειμένου να διευκολυνθεί ο άνθρωπος στην επικοινωνία του με τον υπολογιστή δημιουργήθηκε η Assembly. Η Assembly είναι κωδικοποιημένη γλώσσα μηχανής, δηλαδή υποστηρίζει το ίδιο σύνολο εντολών με τη γλώσσα μηχανής. Οι εντολές της όμως δεν εκφράζονται ως ακολουθίες δυαδικών ψηφίων αλλά παρουσιάζουν κάποια ομοιότητα με τις λέξεις της φυσικής γλώσσας. Για παράδειγμα, η ακολουθία 1011000001100001 συνιστά ορθή εντολή σε γλώσσα μηχανής. Η ίδια εντολή σε Assembly εκφράζεται ως mov al, 061h που σημαίνει μετακίνησε τη δεκαεξαδική τιμή 61 (97 στο δεκαδικό σύστημα) στον καταχωρητή με το όνομα "al". Είναι προφανές πως ο άνθρωπος ευκολότερα ανακαλεί εντολές τύπου Assembly παρά εντολές σε γλώσσα μηχανής.

Ο κώδικας σε Assembly δεν εκτελείται κατευθείαν από τη μηχανή. Πρέπει προηγουμένως να μετατραπεί σε γλώσσα μηχανής. Η μετατροπή αυτή γίνεται από κατάλληλο συμβολομεταφραστή που ονομάζεται Assembler.

Γλώσσες υψηλού επιπέδου: Η Assembly διευκολύνει στην αναγνωσιμότητα των προγραμμάτων αλλά διατηρεί το μειονέκτημα πως περιορίζεται από το σύνολο εντολών της μηχανής. Οι εντολές αυτές αντιστοιχούν σε στοιχειώδεις λειτουργίες της μηχανής με αποτέλεσμα και το πιο απλό πρόγραμμα σε Assembly να παρουσιάζει αυξημένο μέγεθος και πολυπλοκότητα και να απαιτεί υψηλό φόρτο εργασίας.

Αυτός είναι και ο λόγος που η Assembly χαρακτηρίζεται ως γλώσσα χαμηλού επιπέδου (low level). Για παράδειγμα, ο κώδικας που ακολουθεί είναι σε Assembly x86-64 και τυπώνει στην οθόνη τον χαρακτήρα '!'.

```
push    $0x21       # '!'
mov     $1, %rax    # sys_write call number
mov     $1, %rdi    # write to stdout (fd=1)
mov     %rsp, %rsi  # use char on stack
mov     $1, %rdx    # write 1 char
syscall
add     $8, %rsp    # restore sp
```

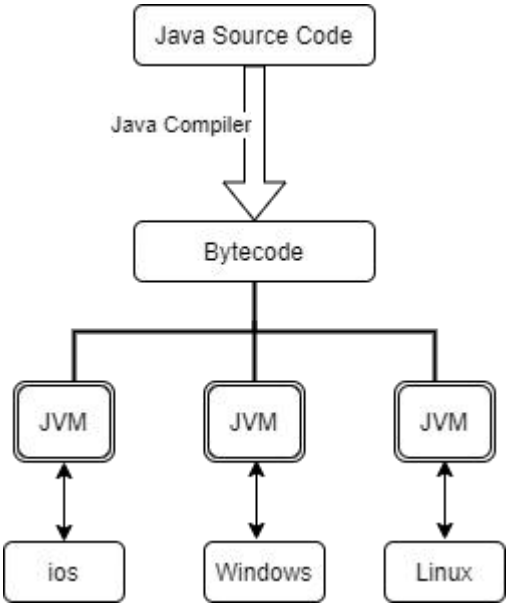
Έτσι προέκυψε η ανάγκη για τις γλώσσες υψηλού επιπέδου (high level programming languages). Ήδη το 1957 κατασκευάστηκε η πρώτη γλώσσα υψηλού επιπέδου, η Fortran της IBM. Οι γλώσσες αυτές είναι ακόμη πιο κοντά στον άνθρωπο σε σχέση με την Assembly και υποστηρίζουν περισσότερες και πιο σύνθετες εντολές. Για παράδειγμα, σε σύγκριση με την Assembly, ο κώδικας Java που εκτυπώνει τον χαρακτήρα '!' είναι System.out.print('!'). Οι γλώσσες υψηλού επιπέδου αυξάνουν δραματικά την παραγωγικότητα κατά την ανάπτυξη εφαρμογών, καθώς ο κώδικας είναι απλούστερος σε σύγκριση με μία γλώσσα μηχανής και Assembly, ενώ δεν απαιτούν να γραφεί διαφορετικό πρόγραμμα για κάθε διαφορετική μηχανή. Ο κώδικας που εκφράζεται σε γλώσσα υψηλού επιπέδου ή Assembly ονομάζεται πηγαίος κώδικας (source code). Σήμερα υπάρχει μεγάλο πλήθος γλωσσών υψηλού επιπέδου με κυριότερους αντιπροσώπους τις C, C++, C#, C objective, Java, Python, Javascript, PHP.

Μεταγλωττιστής: Είναι ευνόητο πως ο πηγαίος κώδικας δεν είναι εκτελέσιμος από τη μηχανή. Ο μεταγλωττιστής (compiler) λαμβάνει ως είσοδο πηγαίο κώδικα και παράγει κώδικα σε γλώσσα μηχανής. Με αυτόν τον τρόπο κωδικοποιείται μια λειτουργία μια φορά σε γλώσσα υψηλού επιπέδου και από αυτήν την κωδικοποίηση μπορεί να παραχθεί εκτελέσιμος κώδικας για κάθε μηχανή που διαθέτει τον αντίστοιχο μεταγλωττιστή. Ένα από τα πλεονεκτήματα του μεταγλωττιστή είναι πως διευκολύνει την προστασία των πνευματικών δικαιωμάτων του κώδικα, καθώς ο κατασκευαστής του λογισμικού δεν διαθέτει στον πελάτη τον πηγαίο αλλά μόνο τον εκτελέσιμο κώδικα. Επιπλέον, διευκολύνει τον προγραμματιστή, καθώς εντοπίζει μια σειρά από συντακτικά κυρίως λάθη κατά τη φάση της μεταγλώττισης (compile time).

Διασυνδέτης: Συχνά, ο πηγαίος κώδικας μίας εφαρμογής αποτελείται από περισσότερα από ένα αρχεία. Κάθε ένα από αυτά τα αρχεία μεταγλωττίζεται χωριστά και για κάθε ένα παράγεται ένα χωριστό αρχείο. Προκειμένου όμως η εφαρμογή να τρέξει, προκύπτει η ανάγκη τα αρχεία που παράγονται από τον μεταγλωττιστή να διασυνδεθούν μεταξύ τους. Την εργασία αυτή αναλαμβάνει ο διασυνδέτης (linker).

Διερμηνευτής: Ο διερμηνευτής (Interpreter) επίσης μεταφράζει τον πηγαίο κώδικα σε εκτελέσιμο αλλά το κάνει κατά τον χρόνο εκτέλεσης (run time), δηλαδή ο διερμηνευτής κατά τον χρόνο εκτέλεσης απαιτεί διαθέσιμο τον πηγαίο κώδικα που μεταφράζει και εκτελεί εντολή προς εντολή. Επομένως ο διερμηνευτής δεν εντοπίζει τα συντακτικά λάθη παρά μόνο κατά τον χρόνο εκτέλεσης του προγράμματος, ενώ δεν παρέχει προστασία των πνευματικών δικαιωμάτων του πηγαίου κώδικα. Επιπλέον, ένα πρόγραμμα σε γλώσσα διερμηνευτή απαιτεί μεγαλύτερο χρόνο εκτέλεσης από το ίδιο πρόγραμμα μεταγλωττισμένο.

Εικονική μηχανή: Όπως αναφέραμε προηγουμένως, ο μεταγλωττιστής μεταφράζει τον πηγαίο κώδικα σε εκτελέσιμο. Πρόκειται για περίπλοκο λογισμικό και η υλοποίησή του απαιτεί αρκετή εργασία και υψηλό κόστος. Όταν μάλιστα θέλουμε τα προγράμματα που αναπτύσσουμε σε μια γλώσσα να είναι εκτελέσιμα σε πολλές πλατφόρμες, είμαστε αναγκασμένοι να υλοποιήσουμε έναν διαφορετικό μεταγλωττιστή για κάθε πλατφόρμα. Μια εναλλακτική λύση είναι η αξιοποίηση εικονικής μηχανής. Στην περίπτωση αυτή έχουμε συνδυασμό ενός μεταγλωττιστή με μια εικονική μηχανή. Ο μεταγλωττιστής όμως δεν παράγει κώδικα εκτελέσιμο απευθείας από τη μηχανή. Αντίθετα, παράγει έναν ενδιάμεσο κώδικα, γνωστό ως bytecode, ο οποίος αποτελεί είσοδο για την εικονική μηχανή που με τη σειρά της παράγει τον εκτελέσιμο κώδικα. Επιπλέον, ο ενδιάμεσος κώδικας δεν διασυνδέεται από έναν τυπικό διασυνδέτη αλλά από τον επονομαζόμενο φορτωτή κλάσης (class loader). Η τεχνολογία αυτή έχει υιοθετηθεί από την Java με την υλοποίηση της Java Virtual Machine ή JVM για συντομία.



Σχήμα 1.1: *Η λειτουργία της εικονικής μηχανής της Java*

Όπως φαίνεται στο Σχήμα 1.1, ο πηγαίος κώδικας Java μεταφράζεται σε Bytecode. Ο κώδικας Bytecode είναι ανεξάρτητος από τη φυσική μηχανή στην οποία θέλουμε να τρέξει. Στη συνέχεια, ο ενδιάμεσος κώδικας τρέχει σε διάφορα λειτουργικά μέσω της εικονικής μηχανής. Επομένως δεν χρειαζόμαστε έναν μεταγλωττιστή για κάθε λειτουργικό σύστημα. Αντίθετα, ένας μεταγλωττιστής για όλα τα λειτουργικά μαζί με μια εικονική μηχανή για κάθε ένα λειτουργικό αρκούν. Η εικονική μηχανή όμως είναι πολύ απλούστερη σε σχέση με τον μεταγλωττιστή και έτσι η ανάπτυξή της για κάθε λειτουργικό είναι πολύ οικονομικότερη από την ανάπτυξη μεταγλωττιστή.

Ο μεταγλωττιστής Just In Time (JIT): Ο μεταγλωττιστής JIT εκτελείται από την εικονική μηχανή και η δουλειά του είναι η μεταγλώττιση του bytecode σε εκτελέσιμο. Ο μεταγλωττιστής JIT δεν ενδιαφέρεται για συντακτικά λάθη καθώς αυτά έχουν εντοπιστεί και διορθωθεί κατά τη μεταγλώττιση του πηγαίου κώδικα. Επιπλέον, δεν μεταφράζει άμεσα όλον τον bytecode σε εκτελέσιμο. Αντίθετα, μεταφράζει μόνο τα τμήματα που είναι αναγκαία ανάλογα με τις κλήσεις που περιλαμβάνει ο κώδικας. Έτσι η εκτέλεση και η μετάφραση του κώδικα γίνονται παράλληλα. Το αποτέλεσμα είναι πως η ταχύτητα εκτέλεσης του bytecode προσεγγίζει την ταχύτητα εκτέλεσης του κώδικα μηχανής. Ας σημειωθεί πως στη βιβλιογραφία ο JIT συχνά αναφέρεται ως μεταγλωττιστής, ενώ κάποιες φορές αναφέρεται ως διερμηνευτής (Java Interpreter). Στην πράξη πρόκειται για κάτι ενδιάμεσο. Ο JIT μετατρέπει τον bytecode σε εκτελέσιμο κατά τον χρόνο εκτέλεσης που είναι χαρακτηριστικό του διερμηνευτή. Από την άλλη πλευρά όμως, ένα τμήμα κώδικα που έχει ήδη μεταφραστεί δεν απαιτείται να μεταφραστεί και πάλι όταν χρησιμοποιείται κατά την τρέχουσα εκτέλεση του προγράμματος για δεύτερη φορά.

JRE και JDK: Το Java Runtime Environment (JRE) περιλαμβάνει την JVM και επιπλέον κάποιες βιβλιοθήκες της Java που είναι απαραίτητες για να εκτελεστεί μια εφαρμογή Java. Το Java Development Kit (JDK) περιλαμβάνει την JRE και μια σειρά από εργαλεία απαραίτητα για την ανάπτυξη πηγαίου κώδικα σε Java, π.χ. ο μεταγλωττιστής που μεταφράζει από πηγαίο κώδικα σε bytecode.

Απολαθοποιητής (Debugger): Ο απολαθοποιητής είναι ένα λογισμικό που μας βοηθάει να εντοπίζουμε λάθη στον κώδικα. Ο απολαθοποιητής μας δίνει τη δυνατότητα να εκτελούμε το πρόγραμμά μας ή κάποιο τμήμα του, βήμα-βήμα, και σε κάθε βήμα να ελέγχουμε το περιεχόμενο των μεταβλητών. Έτσι διευκολυνόμαστε να εντοπίσουμε σε ποιο σημείο ακριβώς συμβαίνει το λάθος και να το διορθώσουμε.

Ολοκληρωμένα Περιβάλλοντα Ανάπτυξης: Τα Ολοκληρωμένα Περιβάλλοντα Ανάπτυξης (Integrated Development Environment ή IDE για συντομία) είναι λογισμικά που σκοπό έχουν να μας βοηθήσουν στην ανάπτυξη εφαρμογών. Σήμερα είναι διαθέσιμα πολλά τέτοια περιβάλλοντα για ποικίλες γλώσσες.

Παραδείγματα αποτελούν το NetBeans, το Eclipse, το Visual Studio, το IntelliJ, το PyCharm κ.ά. Κατά τη συγγραφή αυτού του βιβλίου χρησιμοποιήθηκε το NetBeans, έκδοση 12.5.

Διεπαφή Προγραμματισμού Εφαρμογών: Η Διεπαφή Προγραμματισμού Εφαρμογών (Application Programming Interface ή API για συντομία) είναι μια βιβλιοθήκη κώδικα που μπορεί να χρησιμοποιηθεί για την ανάπτυξη εφαρμογών.

1.3 Βασικές αρχές σχεδίασης της Java

Η Java σχεδιάστηκε στη βάση των ακόλουθων αρχών [8]:

Απλότητα, Αντικειμενοστρέφεια και Οικειότητα
Η Java σχεδιάστηκε να είναι απλή στη χρήση της ώστε οι προγραμματιστές να γίνουν σύντομα παραγωγικοί. Πιθανότατα σε αυτήν την αρχή βασίστηκε το απλό μοντέλο που υιοθετεί η Java στη διαχείριση της μνήμης. Επιπλέον είναι χρήσιμο, οι προγραμματιστές να είναι ήδη εξοικειωμένοι με τις δομές της Java. Έτσι αποφασίστηκε να ομοιάζουν οι δομές της Java με αυτές της C++. Επίσης, η Java να υποστηρίζει ολοκληρωμένα το αντικειμενοστρεφές μοντέλο.

Αξιοπιστία και Ασφάλεια
Η Java σχεδιάστηκε για να υλοποιεί εφαρμογές υψηλής αξιοπιστίας. Τα χαρακτηριστικά της γλώσσας καθοδηγούν τους προγραμματιστές να υιοθετούν πρακτικές που καθιστούν τους κώδικες αξιόπιστους και σταθερούς. Η ασφάλεια αποτέλεσε κρίσιμο ζητούμενο κατά τον σχεδιασμό. Ενσωματωμένα στη γλώσσα χαρακτηριστικά καθιστούν τις εφαρμογές της Java δύσκολο στόχο στις επιθέσεις από κακόβουλο λογισμικό.

Φορητότητα
Οι εφαρμογές σε Java θα πρέπει να εκτελούνται χωρίς πρόβλημα σε οποιαδήποτε πλατφόρμα. Η αρχή αυτή επιτυγχάνεται με την υιοθέτηση της αρχιτεκτονικής της εικονικής μηχανής. Σε οποιαδήποτε πλατφόρμα υπάρχει η JVM, οι εφαρμογές Java εκτελούνται χωρίς πρόβλημα.

Ταχύτητα
Η Java επιτυγχάνει πολύ υψηλές ταχύτητες. Σε αυτό συμβάλλουν πολλά χαρακτηριστικά της γλώσσας όπως η αυτόματη διαχείριση της μνήμης (ενότητα 3.8.1), η έξυπνη μετάφραση του κώδικα από τον JIT αλλά και ο καθορισμός των τύπων των μεταβλητών κατά τον χρόνο μεταγλώττισης (ενότητα 3.2).

Πολυνηματικός και δυναμικός χαρακτήρας και διερμήνευση
Ο διερμηνευτής JIT της Java μπορεί να εκτελέσει κώδικα σε οποιαδήποτε μηχανή είναι εγκατεστημένη η εικονική μηχανή. Η Java σχεδιάστηκε εξαρχής ώστε να υποστηρίζει πολυνηματισμό, δηλαδή τη δυνατότητα, τμήματα του κώδικα να εκτελούνται παράλληλα από τον ίδιο επεξεργαστή. Τέλος, ο δυναμικός χαρακτήρας της Java αναφέρεται στη δυναμική σύνδεση που γίνεται κατά τον χρόνο εκτέλεσης. Σε αυτήν τη φάση ενδέχεται να συνδεθούν, ανάλογα με τις ανάγκες, κώδικες ακόμη και από απομακρυσμένες τοποθεσίες του διαδικτύου.

1.4 Τεχνολογίες της Java

Οι τυπικές γλώσσες προγραμματισμού παρέχουν στον προγραμματιστή μια σειρά από βασικές λειτουργίες όπως είναι η δυνατότητα ανάπτυξης επαναληπτικών δομών, οι δομές επιλογής, η δημιουργία αντικειμένων, η διαχείριση της μνήμης, κλπ. Τις δυνατότητες αυτές προσφέρει και η C++. Όμως κατά τη δημιουργία των σύγχρονων εφαρμογών απαιτούνται και άλλες πέραν των βασικών αυτών δυνατοτήτων, π.χ. ο προγραμματισμός της διεπαφής χρήστη (user interface). Έτσι αν αναπτύσσουμε μια εφαρμογή με C++ και χρειαζόμαστε παραθυρική διεπαφή χρήστη, θα πρέπει να αξιοποιήσουμε κατάλληλες βιβλιοθήκες τρίτων που υποστηρίζουν την ανάπτυξη τέτοιων διεπαφών, καθώς η ίδια η C++ δεν παρέχει αυτήν τη δυνατότητα. Αντίθετα η Java υποστηρίζει στο βασικό πλαίσιο και τον προγραμματισμό της διεπαφής χρήστη και πολλές άλλες δυνατότητες όπως θα δούμε αμέσως παρακάτω. Το σύνολο των βασικών δυνατοτήτων στην Java

θεωρείται ότι συνιστά τον πυρήνα της γλώσσας (Java core) πάνω στον οποίο αναπτύσσονται μια σειρά από εξειδικευμένες τεχνολογίες κατάλληλες για την ανάπτυξη ποικίλων εφαρμογών.

Βάσεις Δεδομένων: Η σύνδεση και επικοινωνία με βάση δεδομένων γίνεται με βιβλιοθήκες ενσωματωμένες στο βασικό πλαίσιο της Java. Ακόμη περαιτέρω, η Java διαθέτει τη δική της Βάση Δεδομένων (Java DB).

Ενσωματωμένα Συστήματα: Με το πλαίσιο Java SE Embedded [9], η Java υποστηρίζει την ανάπτυξη αξιόπιστων και φορητών εφαρμογών υψηλής λειτουργικότητας για τα περισσότερα από τα σύγχρονα ενσωματωμένα συστήματα.

Υπηρεσίες αναγνώρισης και εξουσιοδότησης: Το πλαίσιο αναγνώρισης και εξουσιοδότησης (Java Authentication and Authorization Services) χρησιμοποιείται για να προσδιορίζεται ο χρήστης της εφαρμογής ώστε να εξουσιοδοτούνται να εκτελούν συγκεκριμένες εργασίες μόνο όσοι διαθέτουν κατάλληλα δικαιώματα [10, 11].

Διεθνοποίηση: Με το πλαίσιο διεθνοποίησης (Internationalization) [12] μπορούμε εύκολα να αλλάξουμε τη γλώσσα και άλλα τοπικά χαρακτηριστικά μιας εφαρμογής, π.χ. το ημερολόγιο.

Προσαρμοσμένη δικτύωση: Με την υποστήριξη της προσαρμοσμένης δικτύωσης (Custom Networking) [13], η Java μας δίνει τη δυνατότητα ανάπτυξης εφαρμογών που χρησιμοποιούν και αλληλεπιδρούν με πόρους στο Διαδίκτυο και στον Παγκόσμιο Ιστό.

Διανομή εφαρμογών: Με τη διανομή εφαρμογών (Deployment), η Java μας δίνει τη δυνατότητα να παράγουμε και να διανέμουμε μέσω διαδικτύου εμπλουτισμένες εφαρμογές (Rich Internet Applications ή RIA), δηλαδή εφαρμογές που διαθέτουν τα ισχυρά χαρακτηριστικά των εφαρμογών επιτραπέζιου υπολογιστή αλλά διανέμονται μέσω διαδικτύου [14].

Κανονικές εκφράσεις: Η Java υποστηρίζει πλήρως τις κανονικές εκφράσεις (Regular Expressions). Πρόκειται για συμβολοσειρές που καθορίζουν ένα μοτίβο αναζήτησης [15, 16].

Διεπαφή προγραμματισμού εφαρμογών επεξεργασίας XML εγγράφων: Η διεπαφή αυτή (API for XML Processing) παρέχει στις εφαρμογές της Java τη δυνατότητα εύκολης επεξεργασίας XML εγγράφων.

Διεπαφή προγραμματισμού εφαρμογών email και υπηρεσία διαχείρισης μηνυμάτων: Οι διεπαφές αυτές (JavaMail API και Java Message Service API) παρέχουν τη δυνατότητα ανάπτυξης εφαρμογών διαχείρισης και ανταλλαγής μηνυμάτων.

Πλαίσιο ενεργοποίησης JavaBeans: Το πλαίσιο αυτό (JavaBeans Activation Framework, JAF) παρέχει τη δυνατότητα αναγνώρισης του τύπου των δεδομένων. Για παράδειγμα, ένα πρόγραμμα περιήγησης στο διαδίκτυο, λαμβάνει μια σειρά από δεδομένα που συνιστούν μια εικόνα JPEG. Με το πλαίσιο ενεργοποίησης μπορούμε να αναγνωρίσουμε ότι πρόκειται για εικόνα JPEG και να τη διαχειριστούμε κατάλληλα.

Διεπαφή Servlet: Παρέχει λειτουργίες που συνεργάζονται με έναν εξυπηρετητή διαδικτύου (Web Server) και τον βοηθούν στη διαχείριση των αιτημάτων των πελατών (client) [17].

Σελίδες Εξυπηρετητή: Η τεχνολογία αυτή (JavaServer Pages Technology) παρέχει τη δυνατότητα δυναμικής διαμόρφωσης ιστοσελίδων για τις εφαρμογές του Παγκόσμιου Ιστού (Web Applications).

Βιβλιοθήκη ετικετών: Η βιβλιοθήκη ετικετών (JavaServer Pages Standard Tag Library) παρέχει με τη μορφή απλών ετικετών (tags) βασική λειτουργικότητα για εφαρμογές του Παγκόσμιου Ιστού, π.χ. ετικέτες για επαναληπτικές δομές, για δομές επιλογής, για διαχείριση XML εγγράφων [18].

Τεχνολογία JavaServer Faces Technology: Η τεχνολογία αυτή απλοποιεί την ενσωμάτωση διεπαφών χρήστη σε εφαρμογές του Παγκόσμιου Ιστού [19].

JavaFx: Το πλαίσιο αυτό είναι κατάλληλο για την ανάπτυξη εμπλουτισμένων εφαρμογών διαδικτύου. Ενσωματώνει τη δυνατότητα ανάπτυξης επιτραπέζιων εφαρμογών με παραθυρική διεπαφή [20]. Ας σημειωθεί ότι είναι το νεότερο από τα τρία πλαίσια που διαθέτει η Java για ανάπτυξη παραθυρικών εφαρμογών. Τα άλλα δύο είναι το Abstract ToolKit και το Swing.

Άλλα χαρακτηριστικά της Java: Η Java είναι η βασική γλώσσα ανάπτυξης εφαρμογών Android. Επίσης, η JavaFx σε συνεργασία με την τεχνολογία Gluon μπορεί να κτίσει παραθυρικές εφαρμογές και να τις μεταφέρει στις πλατφόρμες κινητών Android και iOS. Ένα άλλο σημαντικό χαρακτηριστικό της Java είναι η συμβατότητα μεταξύ των εκδόσεων (Backwards compatibility). Πρόκειται για αναγκαίο χαρακτηριστικό προκειμένου οι επιχειρήσεις να επενδύουν για ανάπτυξη σε μια γλώσσα. Επιπλέον, η Java είναι ανοιχτό λογισμικό (Open Source). Αυτό διευκολύνει την ανάπτυξη βιβλιοθηκών τρίτων σε μορφή ανοιχτού κώδικα. Για παράδειγμα, αν κάποιος θελήσει να αναπτύξει μια εφαρμογή οπτικής αναγνώρισης χαρακτήρων (Optical Character Recognition, OCR), μπορεί εύκολα κατάλληλες βιβλιοθήκες σε Java. Αντίστοιχες βιβλιοθήκες σε άλλες γλώσσες κοστίζουν συνήθως μερικές χιλιάδες Ευρώ.

Βιβλιογραφία

[1] Μ. Δερτούζος, Η ΑΝΟΛΟΚΛΗΡΩΤΗ ΕΠΑΝΑΣΤΑΣΗ. Λιβάνης, 2001. Accessed: Sep. 14, 2021. [Online]. Available: https://www.politeianet.gr/books/9789601403755-dertouzos-michalis-libanis-i-anolokliroti-epanastasi-23070

[2] Y. Ilyin, "Digital realities in 2045: future technologies and security issues," Kaspersky daily, 2015. https://www.kaspersky.com/blog/digital-realities-in-2045-future-technologies-and-security-issues/15047/ (accessed Sep. 14, 2021).

[3] S. Walter and M. Kenrick, Java μια εισαγωγή στην επίλυση προβλημάτων και στον Προγραμματισμό, 7η έκδοση. Εκδόσεις Τζιόλα, 2016.

[4] "Java (software platform)," Wikipedia. Apr. 24, 2021. Accessed: Jun. 13, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Java_(software_platform)&oldid=1019658740

[5] "Java (programming language)," Wikipedia. Jun. 10, 2021. Accessed: Jun. 13, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=1027807155

[6] S. J. Vaughan-Nichols, "A Google Android and Java history lesson," ZDNet. https://www.zdnet.com/article/a-google-android-and-java-history-lesson/ (accessed Jun. 13, 2021).

[7] Taniani, "What Is JavaFX?," Social net: Databases Oracle, MySQL, programming SQL, Java, APEX, administration, Jul. 03, 2018. https://oracle-patches.com/en/coding/what-is-javafx (accessed Jun. 13, 2021).

[8] "The Java Language Environment." https://www.oracle.com/java/technologies/introduction-to-java.html (accessed Jun. 13, 2021).

[9] "Oracle Java SE Embedded Overview." https://www.oracle.com/java/technologies/javase-embedded/javase-embedded.html (accessed Jun. 13, 2021).

[10] "Java Platform, Standard Edition Security Developer's Guide." https://docs.oracle.com/javase/9/security/java-authentication-and-authorization-service-jaas1.htm#JSSEC-GUID-76141D9C-03F1-40D8-93D8-DFFC2143A2CA (accessed Jun. 13, 2021).

[11] "JAAS Reference Guide." https://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html#Introduction (accessed Jun. 13, 2021).

[12] "Java SE 8 Internationalization (I18n) Developer Guides." https://docs.oracle.com/javase/8/docs/technotes/guides/intl/index.html (accessed Jun. 13, 2021).

[13] "Trail: Custom Networking (The Java™ Tutorials)." https://docs.oracle.com/javase/tutorial/networking/index.html (accessed Jun. 13, 2021).

[14] "Trail: Deployment (The Java™ Tutorials)." https://docs.oracle.com/javase/tutorial/deployment/index.html (accessed Jun. 13, 2021).

[15] "Java Regular Expressions." https://www.w3schools.com/java/java_regex.asp (accessed Jun. 13, 2021).

[16] A. Srivastava, Java 9 Regular Expressions: A hands-on guide to implement zero-length assertions, back-references, quantifiers, and many more. Packt Publishing, 2017.

[17] "Introduction to Java Servlets," GeeksforGeeks, May 28, 2018. https://www.geeksforgeeks.org/introduction-java-servlets/ (accessed Jun. 14, 2021).

[18] "JavaServer Pages Standard Tag Library." https://www.oracle.com/java/technologies/java-server-tag-library.html (accessed Jun. 14, 2021).

[19] “JavaServer Faces (JSF) Tutorial - Tutorialspoint.” https://www.tutorialspoint.com/jsf/index.htm (accessed Jun. 14, 2021).

[20] The Definitive Guide to Modern Java Clients with JavaFX - Cross-Platform Mobile and Cloud Development | Stephen Chin | Apress. Apress, 2019. Accessed: Jun. 14, 2021. [Online]. Available: https://www.apress.com/gp/book/9781484249253

Κεφάλαιο 2

Σύνοψη
Σε αυτήν την ενότητα μαθαίνουμε καταρχάς πώς να εγκαθιστούμε την Java και το περιβάλλον ανάπτυξης NetBeans. Στη συνέχεια παρουσιάζουμε και αναλύουμε ένα απλό εισαγωγικό πρόγραμμα σε Java. Στο πλαίσιο αυτό γίνεται εισαγωγική παρουσίαση των εννοιών: πακέτο (package), η main ως κύρια είσοδος στο πρόγραμμα, το υποπρόγραμμα ως void συνάρτηση χωρίς παραμέτρους και η βασική έξοδος στην οθόνη. Επιπλέον, παρουσιάζονται οι βασικές συμβάσεις ονοματολογίας.

Προαπαιτούμενη γνώση
Δεν απαιτούνται ειδικές γνώσεις για αυτήν την ενότητα.

Λέξεις κλειδιά
Περιβάλλον ανάπτυξης, Κλάση, Συνάρτηση, Ονοματολογία, Βασική Έξοδος.

2. Εγκατάσταση και το πρώτο πρόγραμμα

Όπως στις περισσότερες γλώσσες προγραμματισμού, έτσι και στην Java, ο πηγαίος κώδικας έχει τη μορφή απλού κειμένου. Επομένως μπορεί κανείς να γράψει πρόγραμμα χρησιμοποιώντας οποιονδήποτε απλό επεξεργαστή κειμένου (text editor). Στη συνέχεια, εφόσον έχει εγκαταστήσει την Java, μπορεί να καλέσει τον μεταγλωττιστή της, να μεταγλωττίσει τον πηγαίο κώδικα και στη συνέχεια να καλέσει την εικονική μηχανή για να το τρέξει. Αυτή εν ολίγοις είναι η διαδικασία που ακολουθείτο μέχρι τις αρχές της δεκαετίας του 2000. Ο κώδικας όμως είναι χρήσιμο να παρουσιάζεται κατά τρόπο που να είναι ευανάγνωστος. Σε αυτό συμβάλλει η κατάλληλη στοίχιση των γραμμών των εντολών αλλά και ο κατάλληλος χρωματισμός των διαφορετικών στοιχείων του κώδικα. Η στοίχιση που απαιτείται είναι αρκετά περίπλοκη, καθώς οφείλει να ακολουθεί τις πολύπλοκες δομές του κώδικα. Αντίστοιχα, περίπλοκος είναι και ο χρωματισμός του κώδικα. Οι απλοί επεξεργαστές κειμένου δεν παρέχουν ούτε τη δυνατότητα αυτόματης στοίχισης ούτε τη δυνατότητα αυτόματου χρωματισμού.

Έτσι, προέκυψε η ανάγκη να αναπτυχθούν περιβάλλοντα εξειδικευμένα για συγγραφή και διαχείριση πηγαίου κώδικα. Σύντομα τα περιβάλλοντα, γνωστά ως Ολοκληρωμένα Περιβάλλοντα Ανάπτυξης (Integrated Development Environments ή IDE), ενσωμάτωσαν πλήθος δυνατοτήτων που σκοπό έχουν να διευκολύνουν την ανάπτυξη του λογισμικού.

Τα περισσότερα σύγχρονα IDE υποστηρίζουν πλήθος από γλώσσες προγραμματισμού. Ωστόσο, συνήθως είναι πιο συγγενικά με κάποιες από αυτές. Η εγκατάστασή τους μπορεί να περιλαμβάνει και την αυτόματη εγκατάσταση των συγγενικών γλωσσών προγραμματισμού. Παρέχουν τη δυνατότητα οργάνωσης τόσο των αρχείων του κώδικα σε έργα (projects) αλλά και της εσωτερικής οργάνωσης κάθε αρχείου κώδικα ξεχωριστά. Επιπλέον, αυτοματοποιούν τις διαδικασίες που απαιτούνται ώστε ο πηγαίος κώδικας να μετατραπεί σε εκτελέσιμο και να τρέξει, ενώ παρέχουν συσκευές εξόδου όπου ο προγραμματιστής μπορεί να δει άμεσα το αποτέλεσμα της εκτέλεσης του κώδικα που έγραψε. Επίσης, τα σύγχρονα IDE διαθέτουν ενσωματωμένους απολαθοποιητές και διευκολύνουν τη διαδικασία απολαθοποίησης.

Σήμερα, διατίθεται πλήθος από IDE. Στα πλαίσια αυτού του συγγράμματος όμως χρησιμοποιούμε το NetBeans, έκδοση 12.5, που είναι το συγγενικό IDE της Java.

Στη συνέχεια αυτού του κεφαλαίου, αφού εγκαταστήσουμε το NetBeans μαζί με την Java, παρουσιάζουμε και αναλύουμε το πρώτο μας πρόγραμμα. Μετά εισάγουμε τον αναγνώστη στην έννοια του απλού υποπρογράμματος ή ρουτίνας, επισημαίνουμε βασικές συμβάσεις ονοματολογίας που πρέπει απαραίτητα να ακολουθούμε κατά τη συγγραφή των προγραμμάτων και τέλος παρέχουμε αναγκαίες βασικές πληροφορίες σε σχέση με την εμφάνιση μηνυμάτων από το πρόγραμμα προς τον χρήστη.

2.1 Εγκατάσταση NetBeans και JDK

Μεταβείτε στην τοποθεσία https://netbeans.apache.org/download/nb125/nb125.html και κατεβάστε τον κατάλληλο installer ανάλογα με το λειτουργικό σύστημά σας. Τρέξτε τον installer. Δεχτείτε τις προτεινόμενες επιλογές κατά τη διάρκεια της εγκατάστασης. Με την ολοκλήρωση της εγκατάστασης θα έχετε διαθέσιμα, το

περιβάλλον ανάπτυξης, την JSE, την JEE, τις HTML και Javascript και την PHP. Θα έχετε έτσι ένα ολοκληρωμένο περιβάλλον ανάπτυξης εφαρμογών. Βρείτε το εικονίδιο του NetBeans στην επιφάνεια εργασίας και τρέξτε το. Σε περίπτωση που έχετε ήδη παλαιότερη έκδοση του NetBeans εγκατεστημένη, θα ερωτηθείτε αν επιθυμείτε να κάνετε εισαγωγή των ρυθμίσεών της στη νέα έκδοση. Απαντήστε κατάλληλα. Εφόσον οι ρυθμίσεις στην προηγούμενη έκδοση είναι ορθές, η εισαγωγή τους θα σας φανεί πολλή χρήσιμη. Ας σημειωθεί πως αυτός ο installer εγκαθιστά την έκδοση NetBeans 1.25 με το JDK 17 που αποτελούν τις τελευταίες εκδόσεις και για τα δύο προϊόντα κατά τον χρόνο συγγραφής αυτού του κειμένου.

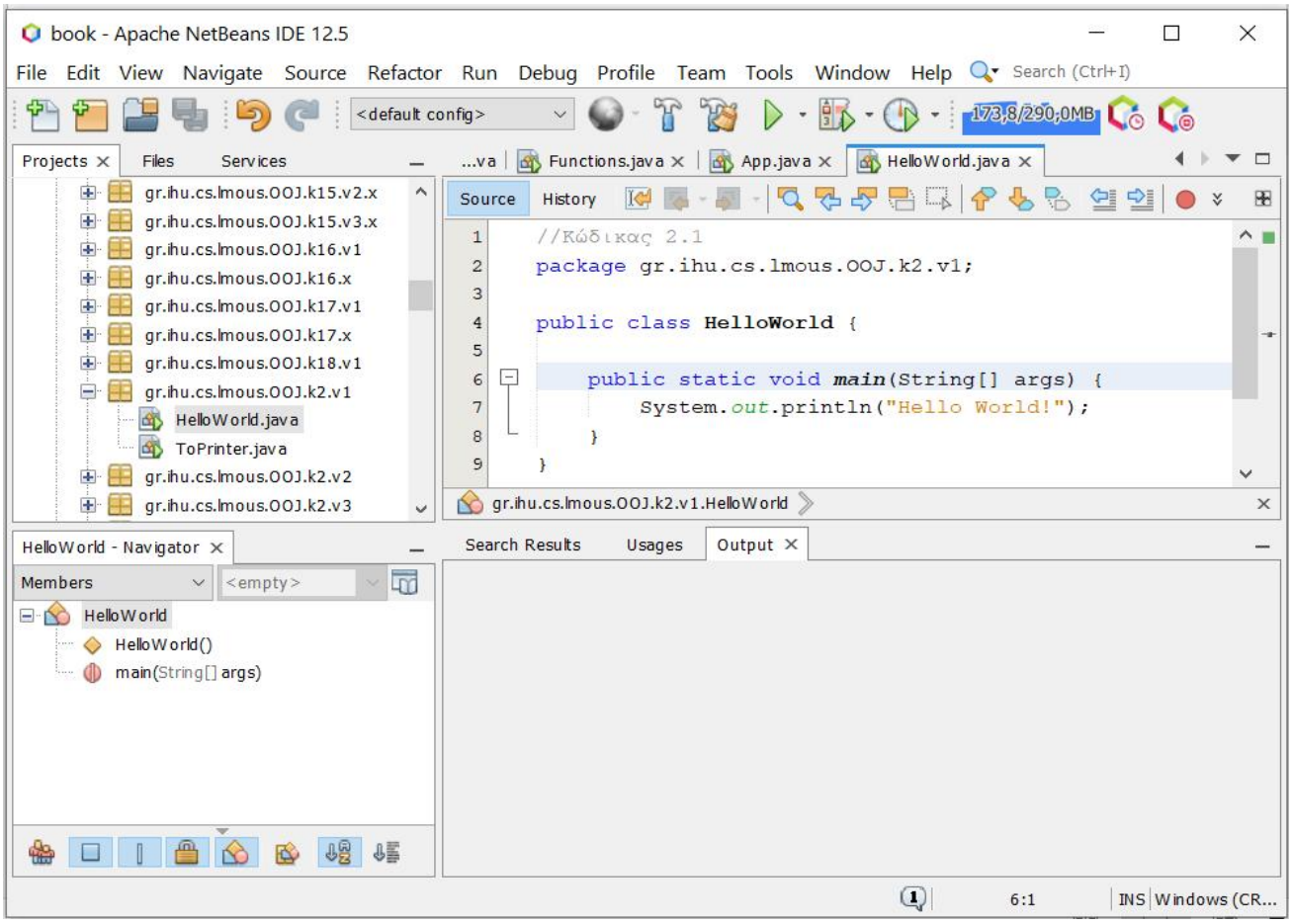
Είστε έτοιμοι να ξεκινήσετε την περιπέτειά σας στον Προγραμματισμό.

2.2 Το πρώτο Πρόγραμμα

Μεταβείτε στο μενού File του NetBeans και επιλέξτε New Project. Στο παράθυρο που θα ανοίξει, επιλέξτε Categories/Java with Ant και Projects/Java Class Library. Πατήστε Next. Στο παράθυρο που θα εμφανιστεί δώστε κατάλληλο όνομα στο project, καθορίστε την τοποθεσία στην οποία θα εμφανιστεί, πατήστε Finish και το project σας έχει δημιουργηθεί. Μπορείτε να καθορίσετε το όνομα της επιλογής σας για το project.

Διαμορφώστε το περιβάλλον εργασίας όπως φαίνεται στην εικόνα 1. Σε αυτήν τη διαμόρφωση περιλαμβάνονται τρία τμήματα. Στα αριστερά είναι τοποθετημένο πλαίσιο με τρία παράθυρα, Projects, Files και Services. Δεξιά επάνω, ο editor, δηλαδή το παράθυρο στο οποίο γράφουμε τον κώδικά μας και δεξιά κάτω το Output, δηλαδή το παράθυρο στο οποίο εμφανίζεται η έξοδος του προγράμματός μας. Αν κάποιο από αυτά τα τμήματα δεν φαίνεται, μεταβείτε στο μενού Window του NetBeans και ανοίξτε το.

Μεταβείτε στο τμήμα Projects, επεκτείνεται το project σας πατώντας στο + που βρίσκεται στα αριστερά του και θα δείτε τους φακέλους του project. Όπως βλέπετε στο project περιλαμβάνονται δύο φάκελοι, ο Source Packages και ο Libraries. Ανοίξτε τον φάκελο Source Packages. Θα διαπιστώσετε πως περιλαμβάνει έναν φάκελο που ονομάζεται Default Package. Ο φάκελος αυτός συνιστά έναν χώρο ή ένα πακέτο όπως λέγεται στο οποίο μπορούμε να τοποθετήσουμε πηγαίους κώδικες. Γενικά τα αρχεία με τους κώδικες τα τοποθετούμε σε πακέτα. Ειδικά όμως η χρήση του Default Package συνίσταται να αποφεύγεται. Αντίθετα, θα πρέπει να δημιουργούμε πακέτα ανάλογα με τις ανάγκες κάθε project. Κάντε δεξί κλικ στο όνομα του project. Από το μενού που θα ανοίξει, επιλέξτε New\Java Package. Δώστε ένα όνομα στο πακέτο και πατήστε finish. Θα διαπιστώσετε πως ο φάκελος Default Package έχει εξαφανισθεί ενώ έχει εμφανισθεί ένας φάκελος με το όνομα που δώσατε στο πακέτο σας. Μέσα σε αυτόν θα τοποθετηθεί το πρώτο πρόγραμμά μας.



***Εικόνα 2.1**: Διαμόρφωση πλαισίου εργασίας:*

Κάντε δεξί κλικ στο πακέτο που δημιουργήσατε, επιλέξτε New\Java Class, ονομάστε την κλάση HelloWorld και πατήστε finish. Προσέξτε πως στο πακέτο σας προστέθηκε ένα αρχείο με το όνομα HelloWorld.java. Παράλληλα, το αρχείο άνοιξε στο τμήμα του editor και είναι έτοιμο για επεξεργασία. Ήδη το NetBeans έχει προσθέσει έναν βασικό κώδικα για να μας βοηθήσει. Πιο συγκεκριμένα, έχει προσθέσει το όνομα του πακέτου στην κορυφή του κώδικα και στη συνέχεια τη δήλωση της κλάσης HelloWorld.

Μετατρέψτε το αρχείο HelloWorld.java όπως δείχνει ο κώδικας 2.1. Αντικαταστήστε όμως το gr.ihu.cs.lmous.OOJ.k1.v1 με το όνομα του δικού σας πακέτου.

```
package gr.ihu.cs.lmous.OOJ.k2.v1;

public class HelloWorld {

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

***Κώδικας 2.1**: Το πρώτο πρόγραμμα*

Αυτός είναι ο πηγαίος κώδικας του πρώτου μας προγράμματος. Στο τμήμα Projects, κάντε δεξί κλικ πάνω στην κλάση HelloWorld και επιλέξτε Run File. Θα εκτελεστεί το πρόγραμμα με αποτέλεσμα να τυπώσει στο παράθυρο Output τη σειρά χαρακτήρων HelloWorld!. Το NetBeans έκανε αυτόματα μεταγλώττιση του κώδικα σε bytecode και ζήτησε από την JVM να τον τρέξει. Αν μεταβείτε στον folder που έχετε αποθηκεύσει το project, θα διαπιστώσετε πως περιέχει μια σειρά από φακέλους. Μεταβείτε στον φάκελο build\classes\ yourPackagePath (Αν χρησιμοποιήσατε το ίδιο όνομα πακέτου με τον κώδικα 2.1, μεταβείτε στον φάκελο

build\classes\gr\ihu\cs\lmous\OOJ\k1\v1), θα δείτε ότι περιέχει ένα αρχείο HelloWorld.class, πρόκειται για το bytecode της κλάσης μας.

Για να τρέξετε την κλάση σας εκτός παραθυρικού περιβάλλοντος, ανοίξτε ένα κέλυφος του λειτουργικού συστήματος, π.χ. στα Windows ανοίξτε ένα powershell ή ένα command prompt. Μεταβείτε στον φάκελο build\classes, πληκτρολογήστε Java yourPackagePath.HelloWorld και Enter, θα δείτε την έξοδο του προγράμματός σας, δηλαδή στο κέλυφος θα εκτυπωθεί η σειρά χαρακτήρων HelloWorld!

Αυτός είναι σε γενικές γραμμές ο τρόπος με τον οποίο θα δημιουργείτε projects, πακέτα και κλάσεις. Τις κλάσεις βέβαια θα τις τρέχετε κυρίως μέσα από το NetBeans καθώς είναι πολύ απλούστερη διαδικασία. Αργότερα, σε αυτό το βιβλίο, θα δούμε πώς να δημιουργούμε εφαρμογές που συνδυάζουν πολλές κλάσεις, οπότε θα δούμε και έναν διαφορετικό τρόπο για να εκτελούμε τις εφαρμογές.

Ας προχωρήσουμε όμως στην ανάλυση αυτού του βασικού κώδικα. Καταρχάς, κάθε κλάση ανήκει σε ένα πακέτο (package). Το όνομα του πακέτου πρέπει να δηλώνεται απαραιτήτως μαζί με τη δεσμευμένη λέξη package στην πρώτη γραμμή της κλάσης. Λεπτομέρειες για τα πακέτα, τη χρησιμότητά τους και την ονοματολογία τους, μπορείτε να βρείτε στην ενότητα 7.8. Στη συνέχεια ακολουθεί η κλάση. Αυτή ορίζεται με χρήση της λέξης-κλειδί class ακολουθούμενη από το όνομα της κλάσης. Στη συνέχεια ανάμεσα σε μια αγκύλη που ανοίγει και μια που κλείνει, τοποθετούμε τον κώδικα της κλάσης. Προσέξτε πως πριν τη λέξη-κλειδί class, έχει τοποθετηθεί η λέξη-κλειδί public. Η λέξη αυτή καθιστά την κλάση δημόσια, δηλαδή οποιοσδήποτε διαθέτει το αρχείο HelloWorld.class μπορεί να χρησιμοποιήσει την κλάση. Η λέξη public είναι ένας προσδιοριστής προσπέλασης (access specifier). Θα μιλήσουμε αναλυτικότερα για τους προσδιοριστές προσπέλασης στην ενότητα 7.7. Θα πρέπει όμως να πούμε πως σε ένα αρχείο .java μπορεί να τοποθετηθεί μια μόνο public κλάση. Είναι υποχρεωτικό μάλιστα, αρχείο και κλάση να έχουν το ίδιο ακριβώς όνομα.

Μέσα στην κλάση ορίζεται μια συνάρτηση που ονομάζεται main. Για την ώρα, μπορούμε να θεωρήσουμε πως μια συνάρτηση είναι ένα υποπρόγραμμα ή αλλιώς μια ρουτίνα [1] που εκτελείται όταν το ζητήσουμε. Όπως θα δούμε αμέσως παρακάτω σε αυτήν την ενότητα, μπορούμε να ορίσουμε πολλές συναρτήσεις μέσα σε μια κλάση οι οποίες θα εκτελούνται μόνο όταν το ζητήσουμε σαφώς. Η συνάρτηση main διαφέρει. Αποτελεί την κύρια είσοδο στο πρόγραμμά μας. Έτσι όταν ζητάμε από την Java να τρέξει μια κλάση, αυτή ψάχνει να βρει τη συνάρτηση main. Αν η κλάση μας δεν περιλαμβάνει main, τότε δεν είναι εκτελέσιμη. Έτσι είτε τρέξουμε την κλάση μας στο NetBeans είτε την τρέξουμε σε κέλυφος του λειτουργικού, η συνάρτηση εκκίνησης είναι η main. Περισσότερα για την main στην ενότητα 7.3.4. Προσέξτε πως η λέξη main ακολουθείται από τη σειρά χαρακτήρων (String[] args). Πρόκειται για τη λίστα παραμέτρων (parameter list) της main. Δεν θα επεκταθούμε περισσότερο σε αυτό για την ώρα. Εκείνο που πρέπει να γνωρίζουμε είναι ότι η main που αποτελεί είσοδο στο πρόγραμμά μας υποχρεωτικά ακολουθείται από τη συγκεκριμένη λίστα παραμέτρων. Περισσότερα για τις λίστες παραμέτρων των συναρτήσεων γενικά στην ενότητα 7.3 και της main ειδικά στην 7.3.4. Μπροστά από την main έχουμε τοποθετήσει τις λέξεις public static void. Το public έχει επίσης την έννοια πως οποιοσδήποτε διαθέτει το bytecode της κλάσης μπορεί να τρέξει τη συνάρτηση. Το static θα το τοποθετούμε μπροστά από όλες τις συναρτήσεις που θα γράφουμε έως ότου εισαχθούμε στο αντικειμενοστρεφές μοντέλο. Τότε θα δούμε μη στατικές συναρτήσεις, δηλαδή συναρτήσεις που δεν χαρακτηρίζουμε ως static και θα διαφοροποιήσουμε με λεπτομέρεια μεταξύ των δύο. Τέλος, το void είναι ο τύπος της τιμής επιστροφής της συνάρτησης. Περισσότερα για τις τιμές επιστροφής των συναρτήσεων στην ενότητα 7.1.

Ο κώδικας που περιέχεται στην main είναι αυτός που στέλνει στην τυπική έξοδο (standard output), δηλαδή στην οθόνη, την σειρά HelloWorld!. Επομένως, αν θέλουμε να τυπώσουμε μια σειρά χαρακτήρων στην οθόνη, μπορούμε να χρησιμοποιούμε την προκαθορισμένη στην Java συνάρτηση System.out.println ακολουθούμενη από παρενθέσεις μέσα στις οποίες τοποθετούμε την σειρά χαρακτήρων περικλείοντάς την σε διπλά εισαγωγικά. Γενικά, οποτεδήποτε αναφερόμαστε μέσα στο πρόγραμμά μας σε σταθερή σειρά χαρακτήρων θα πρέπει να την εσωκλείουμε σε διπλά εισαγωγικά. Βεβαίως, η System.out.println χρησιμοποιείται για την εκτύπωση και άλλου τύπου δεδομένων και όχι μόνο για σειρές χαρακτήρων.

2.3 Οι συναρτήσεις ως απλά υποπρογράμματα

Παρουσιάζουμε εδώ μια εναλλακτική προσέγγιση για την εκτύπωση της σειράς “HelloWorld”. Θα φτιάξουμε μια νέα κλάση που θα ονομάσουμε ToPrinter. Μέσα στην ToPrinter, θα υλοποιήσουμε μια συνάρτηση που θα ονομάσουμε helloWorld. Στη συνέχεια στην main θα καλέσουμε την helloWorld.

Με δεξί κλικ στο πακέτο σας, δημιουργήστε την κλάση ToPrinter. Διαμορφώστε το περιεχόμενο της ToPrinter όπως φαίνεται στον κώδικα 2.2. Σημειώστε πως στο εξής η ονομασία του πακέτου θα παραλείπεται από τους κώδικες σε αυτό το βιβλίο. Ωστόσο, για να είναι οι κώδικες μεταγλωττίσιμοι, θα πρέπει να προσθέτετε εσείς τη δήλωση package συνοδευόμενη από το όνομα του δικού σας πακέτου.

```
package gr.ihu.cs.lmous.OOJ.k2.v1;

public class ToPrinter {

    static void helloWorld() {
        System.out.println("HelloWorld!");
    }

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

***Κώδικας 2.2**: Η πρώτη έκδοση της κλάσης ToPrinter*

Αν τρέξουμε την κλάση, θα λάβουμε το ίδιο αποτέλεσμα με την εκτέλεση της κλάσης HelloWorld. Όπως βλέπουμε στον κώδικα 2.2, η main καλεί την helloWorld() η οποία με τη σειρά της καλεί την System.out.println που τυπώνει τη σειρά "HelloWorld!". Οι συναρτήσεις είναι υποπρογράμματα ή αλλιώς ρουτίνες και αποτελούν θεμέλιο λίθο τόσο του διαδικαστικού όσο και του αντικειμενοστρεφούς μοντέλου και θα τις χρησιμοποιούμε συστηματικά κατά την ανάπτυξη των κλάσεων. Βεβαίως, η συνάρτηση helloWorld αποτελεί την απλούστερη μορφή συνάρτησης. Είναι μια στατική void συνάρτηση χωρίς παραμέτρους. Εκτεταμένα για τις συναρτήσεις μιλάμε στην ενότητα 7. Για την ώρα, μας αρκεί αυτή η απλή μορφή.

Αν υποθέσουμε τώρα πως θέλουμε να εκτυπώσουμε τη διεύθυνσή μας χρησιμοποιώντας την κλάση ToPrinter, μπορούμε να προσθέσουμε μια κατάλληλη συνάρτηση και να την καλέσουμε στην κλάση όπως δείχνει ο κώδικας 2.3.

```
public class ToPrinter {

    static void helloWorld() {
        System.out.println("HelloWorld!");
    }

    static void address() {
        System.out.println("Ιωάννης Νάκος");
        System.out.println("Εσπερίδων 75");
        System.out.println("Καλλιθέα 65303");
        System.out.println("Αθήνα, Ελλάδα");
    }

    public static void main(String[] args) {
        helloWorld();
        address();
    }
}
```

***Κώδικας 2.3**: Η δεύτερη έκδοση της κλάσης ToPrinter*

Προσέξτε όμως ότι θα πρέπει να τοποθετήσετε τη δεύτερη έκδοση της ToPrinter σε διαφορετικό πακέτο από την πρώτη ή διαφορετικά να αλλάξετε την ονομασία της κλάσης. Τρέχοντας τον κώδικα 2.3, διαπιστώνουμε πως τυπώνεται πρώτα η σειρά "HelloWorld" και στη συνέχεια η διεύθυνση. Ένας εύκολος τρόπος για να τυπωθεί μόνο η διεύθυνση είναι να σχολιάσουμε την κλήση της HelloWorld, δηλαδή να μετατρέψουμε την main όπως δείχνει ο κώδικας 2.4.

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

```
        address();
    }
```

***Κώδικας 2.4**: Σχόλια γραμμής*

Όπως βλέπουμε στον κώδικα 2.4, από την κλήση helloWorld() έχουν προηγηθεί δύο ανάστροφοι κάθετοι (backslashes). Με αυτόν τον τρόπο πληροφορούμε τον μεταγλωττιστή πως αυτή η γραμμή είναι γραμμή σχολίων και όχι κώδικα. Τα σχόλια αγνοούνται από τον μεταγλωττιστή. Έτσι, στον bytecode που παράγει ο μεταγλωττιστής δεν συμπεριλαμβάνεται η κλήση helloWorld. Αν εκτελέσουμε τώρα την κλάση ToPrinter, θα διαπιστώσουμε ότι τυπώνει μόνο τη διεύθυνση.

Τα σχόλια είναι γενικά απαραίτητα στον κώδικά μας, όχι μόνο για να αποκλείουμε προσωρινά την κλήση συναρτήσεων αλλά κυρίως για να τοποθετούμε επεξηγήσεις μέσα στον κώδικα ώστε να γίνεται εύκολα κατανοητός από εμάς σε μεταγενέστερο χρόνο αλλά και από συνεργάτες μας που ενδέχεται να δουλέψουν με τον κώδικά μας. Επιπλέον, όπως θα δούμε στην ενότητα 8.6, τα σχόλια αξιοποιούνται και για την παραγωγή της τεκμηρίωσης των κλάσεων. Σε περίπτωση που θέλουμε να εισάγουμε σχόλια που καταλαμβάνουν περισσότερες από μία γραμμές, μπορούμε να τα εσωκλείσουμε μεταξύ των χαρακτήρων ανάστροφη κάθετος και αστερίσκος στην αρχή και αστερίσκος και ανάστροφη κάθετος στο τέλος όπως φαίνεται στον κώδικα 2.5.

```
/*
Στατική συνάρτηση χωρίς παραμέτρους
Τιμή επιστροφής: void
Τυπώνει συγκεκριμένη διεύθυνση
*/
    static void address() {
        System.out.println("Ιωάννης Νάκος");
        System.out.println("Εσπερίδων 75");
        System.out.println("Καλλιθέα 65303");
        System.out.println("Αθήνα, Ελλάδα");
    }
```

***Κώδικας 2.5**: Σχόλια πολλαπλών γραμμών*

Θα προσέξατε πως ο προσδιοριστής προσπέλασης public λείπει από τις δηλώσεις των συναρτήσεων helloWorld και address. Όταν στη δήλωση μιας συνάρτησης δεν τοποθετείται κανένας προσδιοριστής προσπέλασης, η συνάρτηση διατηρεί την εξ' ορισμού προσπέλαση (default access). Η εξ' ορισμού προσπέλαση ονομάζεται και προσπέλαση πακέτου (package access) ή και ιδιωτική προσπέλαση πακέτου (package-private), καθώς αυτή επιτρέπει την προσπέλαση μόνο από κλάσεις του ίδιου πακέτου. Στο πακέτο που τοποθετήσατε τη δεύτερη έκδοση της ToPrinter, δημιουργήστε μια κλάση CallFromAnotherClass. Η κλάση CallFromAnotherClass έχει δικαίωμα προσπέλασης στις συναρτήσεις της ToPrinter. Αντίθετα κλάσεις από άλλα πακέτα δεν έχουν αυτό το δικαίωμα. Για να κληθεί όμως μια συνάρτηση της ToPrinter από την CallFromAnotherClass θα πρέπει κατά την κλήση να προσδιορίσουμε πέραν του ονόματος της συνάρτησης και την κλάση στην οποία αυτή ανήκει.
Ο κώδικας 2. 6. παρουσιάζει την κλάση CallFromAnotherClass

```
public class CallFromAnotherClass {
    public static void main(String[] args) {
        ToPrinter.helloWorld();
        ToPrinter.address();
    }
}
```

***Κώδικας 2.6**: Κλήση στατικής συνάρτησης με πρόσβαση πακέτου από άλλη κλάση του ίδιου πακέτου*

Τρέξτε την CallFromAnotherClass και θα διαπιστώσετε πως θα εκτελεστούν οι συναρτήσεις address και helloWorld της κλάσης ToPrinter. Αν θέλουμε να δώσουμε δικαίωμα κλήσης αυτών των συναρτήσεων από κλάσεις έξω από το πακέτο που ανήκουν θα πρέπει να χαρακτηρίσουμε τις συναρτήσεις ως public.

2.4 Συμβάσεις ονοματολογίας

Η Java διαφοροποιεί μεταξύ πεζών και κεφαλαίων (case sensitive). Αν για παράδειγμα προσπαθήσετε να καλέσετε τη συνάρτηση helloWorld και τη γράψετε ως helloworld, η Java δεν θα την αναγνωρίσει. Για την ονομασία των κλάσεων, των συναρτήσεων αλλά και άλλων αντικειμένων έχουν καθιερωθεί συμβάσεις ονοματολογίας με παγκόσμια ισχύ που θα πρέπει να ακολουθούνται πιστά. Ο μεταγλωττιστής δεν θα μας ενοχλήσει αν παραβούμε μια τέτοια σύμβαση. Ωστόσο, η τήρησή τους είναι αναγκαία ώστε οι κώδικές μας να είναι αναγνώσιμοι από εμάς ή από τρίτους. Πιο συγκεκριμένα, τα ονόματα των κλάσεων πρέπει να αρχίζουν με κεφαλαίο. Αντίθετα, τα ονόματα των συναρτήσεων πρέπει να αρχίζουν από πεζό. Και στις δύο περιπτώσεις, αν το όνομα είναι σύνθετο, π.χ. helloWorld, κάθε επόμενο συνθετικό πρέπει να αρχίζει από κεφαλαίο. Πιο εκτεταμένα για το θέμα αυτό μιλάμε στην ενότητα 3.4 όπου παρουσιάζουμε τους χαρακτήρες που είναι διαθέσιμοι προς χρήση για τη διαμόρφωση των ονομάτων κλάσεων, συναρτήσεων και άλλων αναγνωριστικών (identifier) που χρησιμοποιούμε στα προγράμματά μας.

2.5 Βασική Έξοδος

Είδαμε προηγουμένως πως με την System.out.println μπορούμε να τυπώσουμε μια σειρά χαρακτήρων στην οθόνη. Θα προσέξατε πως κάθε γραμμή της διεύθυνσης τυπώνεται σε διαφορετική γραμμή της οθόνης. Αυτό οφείλεται στην println, η οποία, αφού εμφανίζει τη σειρά χαρακτήρων που της έχουμε δώσει, προετοιμάζει την έξοδο έτσι ώστε αυτή να συνεχίσει σε νέα γραμμή. Οι χαρακτήρες ln στο τέλος της println αυτόν τον σκοπό εξυπηρετούν. Αν αυτό είναι κάτι που δεν θέλουμε, τότε αντί για println θα χρησιμοποιήσουμε τη συνάρτηση print.

Προσθέστε τη συνάρτηση prtABC (κώδικας 2.7) στην ToPrinter, καλέστε την στην main και τρέξτε την κλάση. Η νέα έκδοση της κλάσης φαίνεται στον κώδικα 2.7.

```
public class ToPrinter {

    static void helloWorld() {
        System.out.println("HelloWorld!");
    }

    static void address() {
        System.out.println("Ιωάννης Νάκος");
        System.out.println("Εσπερίδων 75");
        System.out.println("Καλλιθέα 65303");
        System.out.println("Αθήνα, Ελλάδα");
    }

    static void prtABC() {
        System.out.print("A");
        System.out.print("B");
        System.out.println("C");
    }

    public static void main(String[] args) {
        prtABC();
        helloWorld();
    }
}
```

***Κώδικας 2.7**: print και println*

Παρατηρήστε πως στην τελευταία γραμμή της prtABC έχουμε χρησιμοποιήσει println και όχι print. Επιλέξαμε αυτήν την προσέγγιση ώστε οποιαδήποτε έξοδος μετά την κλήση της prtABC να εμφανιστεί σε νέα γραμμή. Έτσι ο κώδικας 2.7 εμφανίζει το HelloWorld! κάτω από τη σειρά ABC. Αν δεν είχαμε βάλει println στην τελευταία print της prtABC, τότε ο κώδικας θα εμφάνιζε ABCHelloWorld!. Περισσότερα για την έξοδο και είσοδο στις ενότητες 4.2 και 4.3.

2.6 Ασκήσεις

Δημιουργήστε μια κλάση X01. Σε αυτήν προσθέστε τις συναρτήσεις:

prtRomvos: Τυπώνει έναν ρόμβο που σχηματίζεται από αστερίσκους όπως φαίνεται παρακάτω:

```
       ***
    *********
***************
***************
    *********
       ***
```

prtVellos: Τυπώνει ένα βέλος όπως φαίνεται παρακάτω:

```
                          *****
                           *****
                            *****
                             *****
********************************
********************************
********************************
                             *****
                            *****
                           *****
                          *****
```

Τρέξτε τις συναρτήσεις και διαπιστώστε πως λειτουργούν σωστά.

Βιβλιογραφία

[1] “Subroutine,” Wikipedia. Sep. 12, 2021. Accessed: Sep. 14, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Subroutine&oldid=1043969790

Κεφάλαιο 3

Σύνοψη
Σε αυτήν την ενότητα εξετάζεται διεξοδικά η έννοια της τοπικής μεταβλητής και αναλύονται σε βάθος οι θεμελιώδεις τύποι δεδομένων της Java. Περιλαμβάνεται παρουσίαση των ακολουθιών ελέγχου, των συμβάσεων ονοματολογίας, του φαινομένου υπερχείλισης μεταβλητής. Εξηγείται η έκφραση αριθμητικών σταθερών στο δυαδικό, οκταδικό και δεκαεξαδικό σύστημα αρίθμησης καθώς και ο επιστημονικός συμβολισμός τους. Επιπλέον, γίνεται εισαγωγή στον βασικό τύπο διαχείρισης αλφαριθμητικών δεδομένων.

Προαπαιτούμενη γνώση
Η ενότητα 2 αυτού του εγχειριδίου που περιλαμβάνει τις έννοιες: πακέτο (package), η main ως κύρια είσοδος στο πρόγραμμα, το υποπρόγραμμα ως void συνάρτηση χωρίς παραμέτρους, βασική έξοδος στην οθόνη.

Λέξεις κλειδιά
Τοπική μεταβλητή (local variable), τοπική σταθερά (local constant), μπλοκ κώδικα (block), εμβέλεια (scope), χρόνος ζωής μεταβλητής (variable life time), αναγνωριστικό (identifier), κυριολεκτική σταθερά (literal), αρχικοποίηση (initialization), χρόνος μεταγλώττισης (Compile time), χρόνος εκτέλεσης (Run time), δεσμευμένη λέξη (reserved word), λέξη κλειδί (keyword), θεμελιώδεις τύποι δεδομένων (Primitive Data Types), κλάση (Class), εκθετική μορφή (exponential Notation) ή επιστημονικός συμβολισμός (scientific notation), αριθμός κινητής υποδιαστολής (floating point number), ακολουθία ελέγχου (control sequence), χαρακτήρας διαφυγής (escape character), υπερχείλιση (overflow), λάθος χρόνου εκτέλεσης (run time error ή bug), λάθος μεταγλώττισης (compile error), αυτόματη μετατροπή τύπου (automatic type conversion) , αλφαριθμητική σειρά (String), σύνδεση αλφαριθμητικών σταθερών (concatenation).

3 Μεταβλητές και Θεμελιώδεις τύποι

Αναφέραμε προηγουμένως πως ένα πρόγραμμα είναι μια σειρά εντολών προς τη μηχανή. Αυτό είναι σωστό, ωστόσο πρόκειται για τη μισή αλήθεια, καθώς οι εντολές συχνά επενεργούν σε δεδομένα. Επομένως, ένα πρόγραμμα για να εκτελεστεί θα πρέπει να φορτωθούν στη μνήμη του υπολογιστή, όχι μόνο οι εντολές που το απαρτίζουν αλλά και τα δεδομένα στα οποία επενεργούν οι εντολές. Το πρόγραμμα φορτώνει τα δεδομένα σε κατάλληλες θέσεις στη μνήμη προς τις οποίες διατηρεί εύκολη πρόσβαση και από εκεί τα διαχειρίζεται. Η πρόσβαση στις θέσεις μνήμης που αντιστοιχούν σε δεδομένα γίνεται μέσω των μεταβλητών (variables).

Ωστόσο, υπάρχουν δεδομένα διάφορων τύπων. Για παράδειγμα, κάποια δεδομένα μπορεί να είναι ακέραιοι αριθμοί, κάποια άλλα μπορεί να είναι πραγματικοί αριθμοί ή χαρακτήρες. Όμως τα δεδομένα αποθηκεύονται στη μνήμη ως σειρές από bits. Επομένως, το πρόγραμμα θα πρέπει να γνωρίζει η κάθε σειρά από bits που αντιστοιχεί σε ορισμένο δεδομένο, τι τύπου είναι ώστε να μπορέσει να τη διαχειριστεί σωστά. Έτσι οι μεταβλητές χαρακτηρίζονται ως προς τον τύπο των δεδομένων που διαχειρίζονται, δηλαδή υπάρχουν μεταβλητές κατάλληλες για ακέραιους, μεταβλητές κατάλληλες για πραγματικούς, κοκ.

Στο κεφάλαιο αυτό παρουσιάζουμε καταρχάς τις τοπικές μεταβλητές και κάποια βασικά χαρακτηριστικά τους όπως η εμβέλεια και ο χρόνος ζωής, στη συνέχεια παρουσιάζουμε τους θεμελιώδεις τύπους της Java. Μετά εξηγούμε τη σημασία και χρήση των ακολουθιών ελέγχου και περιγράφουμε τα αποδεκτά ονόματα μεταβλητών. Ακολουθεί, η επεξήγηση της υπερχείλισης και η έκφραση αριθμητικών σταθερών σε όλα τα συστήματα αρίθμησης που υποστηρίζονται. Τέλος, παρουσιάζεται ο τύπος αλφαριθμητικής σταθεράς ο οποίος δεν είναι μεν θεμελιώδης αλλά το ίδιο βασικός και συχνά αναγκαίος όσο και οι θεμελιώδεις τύποι.

3.1 Τοπικές Μεταβλητές

Ένα πρόγραμμα χρειάζεται να διαχειριστεί κάποια δεδομένα. Η διαχείριση των δεδομένων επιτυγχάνεται με τη βοήθεια των μεταβλητών (variables). Η Java υποστηρίζει μια ποικιλία μεταβλητών, τις τοπικές μεταβλητές

(local variables), τις παραμέτρους (parameters), τις στατικές μεταβλητές (class variables ή static fields) και τις μεταβλητές στιγμιότυπου (instance variables)[1] .

Η συζήτηση στην ενότητα αυτή θα περιοριστεί στις τοπικές μεταβλητές. Οι παράμετροι και οι στατικές μεταβλητές παρουσιάζονται στις ενότητες 7.3 και 7.4, αντίστοιχα, ενώ οι μεταβλητές στιγμιότυπου αφορούν το αντικειμενοστρεφές μοντέλο και εξετάζονται στην ενότητα 11.

Για να δηλώσουμε μια τοπική μεταβλητή[2] στην Java, πρέπει να προσδιορίσουμε τον τύπο της και να της δώσουμε ένα αναγνωριστικό όνομα. Για παράδειγμα, ο κώδικας

```
int k;
```

δηλώνει μια ακέραιη μεταβλητή με αναγνωριστικό το k.

Ας δούμε όμως μερικούς αναγκαίους ορισμούς.

Μπλοκ κώδικα: Ο κώδικας μεταξύ μιας αγκύλης που ανοίγει '{' και μιας αγκύλης που κλείνει '}' συνιστά ένα μπλοκ (block) κώδικα.

Κάθε μεταβλητή που δηλώνεται μέσα σε ένα μπλοκ κώδικα είναι τοπική σε αυτό το μπλοκ.

```
void f() {
     int k;
}
```

Στη συνάρτηση f(), έχουμε δηλώσει την τοπική μεταβλητή ακέραιου τύπου k. Ο int είναι ένας θεμελιώδης τύπος της Java. Οι θεμελιώδεις τύποι των μεταβλητών παρουσιάζονται παρακάτω στην ομώνυμη ενότητα. Για την ώρα μας ενδιαφέρουν τα ακόλουθα:

Η εκτέλεση της int k; έχει ως αποτέλεσμα να δεσμευτεί χώρος στη μνήμη που έχει εκχωρηθεί στο πρόγραμμά μας, με μέγεθος ίσο με το μέγεθος μια ακέραιας μεταβλητής. Ο χώρος αυτός συνδέεται με το αναγνωριστικό (identifier) k. Η μεταβλητή k είναι τοπική στο μπλοκ που ορίζει την f. Κάθε φορά που μέσα στην f αναφερόμαστε στην k, στην ουσία προσπελαύνουμε τη συγκεκριμένη μνήμη. Ωστόσο, δεν έχουμε δώσει κάποια αρχική τιμή στην k. Με άλλα λόγια, η k είναι αναρχικοποίητη (unitialized)[3]. Προκειμένου να μας προστατεύσει από πιθανά λάθη που προκύπτουν από τη χρήση αναρχικοποίητων μεταβλητών, η Java δεν μας επιτρέπει να την προσπελάσουμε παρά μόνο για να την αρχικοποιήσουμε[4]. Έτσι ο παρακάτω κώδικας παράγει ένα λάθος μεταγλώττισης (Compile Error).

```
void f() {
     int k;
     System.out.println(k);
}
```

Αντίθετα, ο κώδικας

```
void f() {
```

---

[1] *Επιπλέον, οι μεταβλητές ανάλογα με τον τύπο τους μπορεί να είναι μεταβλητές αναφοράς (reference variable) ή* ***μεταβλητές τιμής*** *(value variable).*

[2] *Πριν τον τύπο μπορεί να τοποθετηθούν τροποποιητές (modifiers). Στους πιθανούς τροποποιητές περιλαμβάνονται οι* ***προσδιοριστές προσπέλασης*** *(access specifer), η δεσμευμένη λέξη* ***static*** *και η δεσμευμένη λέξη* ***final****. Από αυτούς τους τροποποιητές μόνο ο* ***final*** *συντάσσεται με δήλωση τοπικών μεταβλητών.*

[3] *Η Java δεν αρχικοποιεί αυτόματα τις τοπικές μεταβλητές. Δεν ισχύει το ίδιο για τους υπόλοιπους τύπους μεταβλητών.*

[4] *Μπορούμε ωστόσο να προσπελάσουμε μια μεταβλητή που ενδεχομένως είναι αναρχικοποίητη εφόσον την περικλείσουμε σε try {} catch μπλοκ. Περισσότερα επ'αυτού κατά τη μελέτη των εξαιρέσεων, ενότητα ?*

```
        int k;
        k = 0;
        System.out.println(k);
}
```

περνάει μεταγλώττιση και όταν εκτελεστεί τυπώνει στην οθόνη το περιεχόμενο της k, δηλαδή το 0.

Εμβέλεια: Εμβέλεια (Scope) μιας μεταβλητής ονομάζεται η περιοχή εκείνη του κώδικα που η μεταβλητή είναι αναγνωρίσιμη από τον μεταγλωττιστή.

Η εμβέλεια των τοπικών μεταβλητών αρχίζει από τη γραμμή στην οποία δηλώθηκαν και επεκτείνεται σε όλο το μπλοκ μέσα στο οποίο δηλώθηκαν. Στον κώδικα που ακολουθεί, ο μεταγλωττιστής θα παράξει λάθος καθώς προσπαθούμε να προσπελάσουμε τη μεταβλητή i έξω από το μπλοκ στο οποίο δηλώθηκε.

```
void f() {
        int k = 0;
        System.out.println(k);
        {
                int i = 1;
        }
        System.out.println(i);
}
```

Προσέξτε πως σε αυτό το παράδειγμα έχουμε εμφωλιασμένα (nested) μπλοκ, δηλαδή έχουμε το μπλοκ της f και μέσα σε αυτό, ένα εσωτερικό μπλοκ μέσα στο οποίο δηλώνεται η i. Η εμβέλεια επομένως της i περιορίζεται στο εσωτερικό μπλοκ. Γενικά, η εμβέλεια των τοπικών μεταβλητών περιορίζεται μέσα στο μπλοκ που δηλώθηκαν. Αυτός είναι ένας λόγος στον οποίο οφείλεται ο χαρακτηρισμός τους ως τοπικές. Ένας δεύτερος λόγος σχετίζεται με τον χρόνο ζωής τους.

Χρόνος ζωής: Ο χρόνος ζωής (variable life time) μιας μεταβλητής είναι το διάστημα από τη δημιουργία της μεταβλητής με τη δέσμευση της κατάλληλης μνήμης μέχρι την καταστροφή της με την αποδέσμευση της μνήμης.

Για παράδειγμα, κατά την εκτέλεση της πρώτης γραμμής της f, δεσμεύεται χώρος στη μνήμη και έχουμε την έναρξη της ζωής της μεταβλητής. Με το τέλος εκτέλεσης της f(), ο χώρος για την k αποδεσμεύεται και έχουμε τη λήξη της ζωής της. Στο παράδειγμα αυτό η εμβέλεια μοιάζει να είναι παρόμοια έννοια με τον χρόνο ζωής. Ωστόσο, εμβέλεια και χρόνος ζωής συνιστούν δύο εντελώς διαφορετικές έννοιες. Η μεν εμβέλεια ορίζεται στο πλαίσιο του χρόνου μεταγλώττισης (Compile time), ο δε χρόνος ζωής ορίζεται στο πλαίσιο του χρόνου εκτέλεσης (Run time). Στην ενότητα 11.5 διαφοροποιείται πληρέστερα η εμβέλεια από τον χρόνο ζωής.

Η Java δεν επιτρέπει τη δήλωση μεταβλητής σε εσωτερικό μπλοκ με αναγνωριστικό που έχει χρησιμοποιηθεί σε αντίστοιχο εξωτερικό. Για παράδειγμα, ο κώδικας

```
void f() {
        int k = 0;
        {
                int k = 1;
        }
}
```

θα παράξει λάθος μεταγλώττισης.
Ας σημειωθεί ότι δήλωση και αρχικοποίηση μιας ή περισσότερων μεταβλητών μπορεί να γίνει στην ίδια γραμμή κώδικα, π.χ.

```
int k=0;
```

```
int k=1, z=2;
int k=2, z;
```

Στην πρώτη περίπτωση δηλώνουμε και αρχικοποιούμε την k, στη δεύτερη δηλώνουμε και αρχικοποιούμε τις k και z και στην τρίτη δηλώνουμε τις k και z, αρχικοποιούμε όμως μόνο την k.

3.1.2 Τοπικές Σταθερές

Σε πολλές περιπτώσεις χρειαζόμαστε μεταβλητές που η τιμή τους δεν μπορεί να αλλάξει μετά την αρχική εκχώρηση. Οι μεταβλητές αυτού του τύπου ονομάζονται σταθερές (constant variables). Μια τοπική μεταβλητή μετατρέπεται σε σταθερά αν κατά τη δήλωσή της χρησιμοποιηθεί η δεσμευμένη λέξη (reserved word) final. Για παράδειγμα,

```
final int K=1;
```

Ο κώδικας αυτός μέσα σε ένα μπλοκ δηλώνει μια τοπική σταθερά. Σε μια τοπική σταθερά μπορεί να εκχωρηθεί τιμή αυστηρά μία φορά, είτε μαζί με τη δήλωσή της είτε αργότερα. Επομένως, ο κώδικας

```
final int K;
K=1;
```

είναι έγκυρος.
Αντίθετα, ο κώδικας

```
final int K=1;
K=2;
```

παράγει λάθος μεταγλώττισης.

Όπως φαίνεται από τα παραδείγματα αυτής της ενότητας, οι συμβάσεις ονοματολογίας για τις σταθερές διαφέρουν από τις αντίστοιχες συμβάσεις για μεταβλητές. Οι σταθερές ονομάζονται αποκλειστικά με χρήση κεφαλαίων χαρακτήρων.

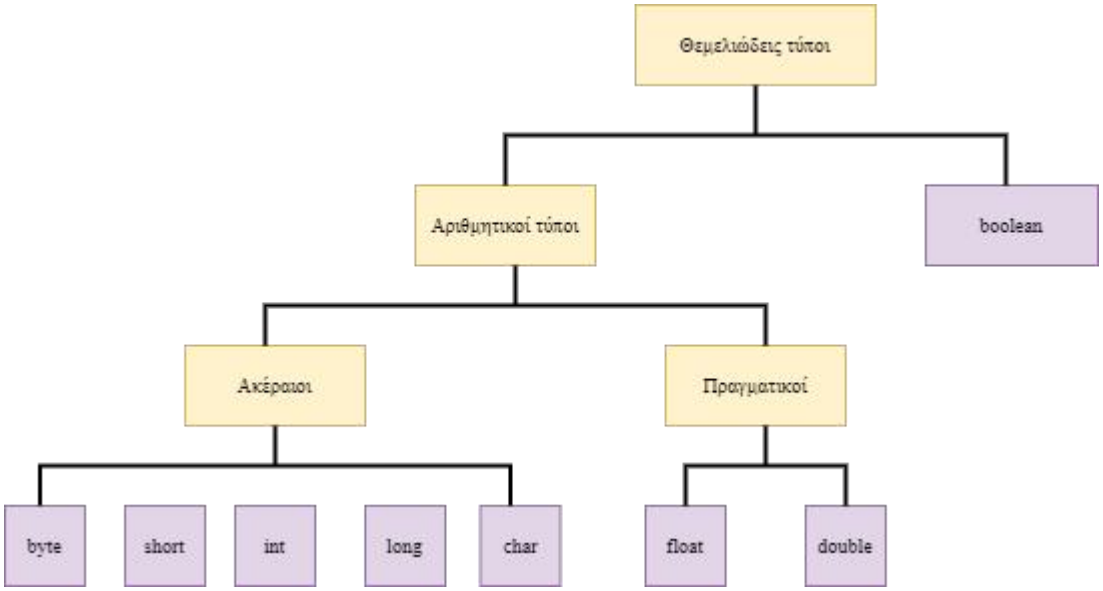
3.2 Θεμελιώδεις τύποι δεδομένων

Όπως εξηγήσαμε στην προηγούμενη ενότητα, τα προγράμματα διαχειρίζονται τα δεδομένα τους με τη βοήθεια των μεταβλητών. Τα δεδομένα είναι διάφορων τύπων, π.χ. ακέραιοι αριθμοί, πραγματικοί, χαρακτήρες, κλπ. Αντίστοιχα, και οι μεταβλητές που χρησιμοποιούνται τυποποιούνται ως ακέραιες, πραγματικές, μεταβλητές χαρακτήρων, κλπ. Η Java θέτει αυστηρούς κανόνες σχετικά με τον προσδιορισμό τύπου μιας μεταβλητής. Πιο συγκεκριμένα, απαιτεί ο προσδιορισμός τύπου κάθε μεταβλητής να είναι δεδομένος κατά τον χρόνο μεταγλώττισης και να παραμένει αμετάβλητος για όλη τη ζωή της μεταβλητής. Γλώσσες με παρόμοια συμπεριφορά χαρακτηρίζονται ως γλώσσες ισχυρού τύπου (Strong typed languages), σε αντίθεση με τις γλώσσες που δεν θέτουν αυτές τις απαιτήσεις και χαρακτηρίζονται ως γλώσσες αδύναμου τύπου (Weak typed languages).

Ένα επίσης σημαντικό χαρακτηριστικό της Java είναι η πρώιμη διασύνδεση (early binding). Όπως αναφέραμε μια μεταβλητή μπορεί να φιλοξενεί μια ακέραιη τιμή ή έναν χαρακτήρα. Και στις δύο περιπτώσεις όμως τα δεδομένα στη μνήμη αποθηκεύονται ως μια σειρά από δυαδικά ψηφία. Για να είναι δυνατή η ορθή ερμηνεία αυτών των δεδομένων είναι αναγκαίο να γνωρίζουμε τον τύπο της μεταβλητής. Με άλλα λόγια, όταν προσπελαύνουμε μια θέση μνήμης, δηλαδή μια σειρά από bits, πρέπει να είναι γνωστό αν η σειρά αυτή αναπαριστά έναν ακέραιο ή έναν χαρακτήρα. Η Java χαρακτηρίζει κάθε μεταβλητή με συγκεκριμένο τύπο, γνωστό κατά τον χρόνο μεταγλώττισης και σταθερό καθόλη την εκτέλεση του προγράμματος. Έτσι όταν προσπελαύνει μια μεταβλητή γνωρίζοντας τη διεύθυνσή της στη μνήμη και τον τύπο της μπορεί άμεσα να την αξιοποιήσει. Στον αντίποδα, οι γλώσσες καθυστερημένης διασύνδεσης (late binding), επιτρέπουν κατά τη διάρκεια εκτέλεσης του προγράμματος την εκχώρηση στην ίδια μεταβλητή δεδομένων ποικίλων τύπων. Για παράδειγμα, η Javascript, επιτρέπει την εκχώρηση σε μεταβλητή μιας ακέραιας τιμής και παρακάτω στον ίδιο κώδικα την εκχώρηση μιας συμβολοσειράς. Επομένως, ο τύπος της μεταβλητής δεν μπορεί να καθοριστεί

παρά μόνο κατά τον χρόνο εκτέλεσης. Έτσι όταν η Javascript προσπελαύνει μια μεταβλητή θα πρέπει να υπολογίσει ποιος είναι ο τύπος της σε ακριβώς εκείνο το σημείο της εκτέλεσης. Όπως είναι φανερό, η πρώιμη διασύνδεση βελτιώνει την ταχύτητα εκτέλεσης.

Οι μεταβλητές επομένως στην Java έχουν προκαθορισμένο τύπο. Οι διαθέσιμοι τύποι είναι πάρα πολλοί. Επιπλέον, ο χρήστης έχει τη δυνατότητα να ορίσει και δικούς του τύπους (User Defined Types). Σε αυτήν την ενότητα όμως παρουσιάζουμε ειδικά τους θεμελιώδεις τύπους (primitive types) της Java. Οι θεμελιώδεις τύποι είναι οκτώ [1], είναι προκαθορισμένοι (predefined) και τα ονόματά τους αποτελούν δεσμευμένες λέξεις (reserved words). Πλήρη λίστα με τις λέξεις κλειδιά της Java μπορείτε να βρείτε στο παράρτημα 2. Όλοι οι υπόλοιποι τύποι προκύπτουν από συνδυασμό ή επανορισμό των θεμελιωδών τύπων. Στο σχήμα 3.1, παρουσιάζεται η κατηγοριοποίηση των θεμελιωδών τύπων.



***Σχήμα 3.1**: Κατηγοριοποίηση των θεμελιωδών τύπων*

byte Ο τύπος byte είναι προσημασμένος ακέραιος μήκους οκτώ δυαδικών ψηφίων (bits). Αν η τιμή μιας μεταβλητής τύπου byte είναι θετική, τότε αναπαρίσταται στη μνήμη κωδικοποιημένη με τη μορφή πρόσημο και μέτρο (sign and magnitude), δηλαδή το πρώτο bit έχει την τιμή 0 για να δείξει πως ο αριθμός είναι θετικός και τα υπόλοιπα bits διατηρούν την τιμή του αριθμού εκφρασμένη στο δυαδικό σύστημα. Αν η τιμή μιας μεταβλητής τύπου byte είναι αρνητική, αυτή αναπαρίσταται στη μνήμη με τη μορφή συμπληρώματος ως προς 2 (two's complement) (παράρτημα 1). Σε αυτήν την περίπτωση το πρώτο bit έχει την τιμή 1 που δείχνει πως ο αριθμός είναι αρνητικός. Επομένως, τόσο για τους θετικούς όσο και για τους αρνητικούς, εξαιρώντας το πρώτο bit που ονομάζεται και bit πρόσημου, διαπιστώνουμε πως για την αναπαράσταση της τιμής απομένουν 7 bits. Συνεπώς, αναπαρίστανται συνολικά 27 αρνητικές τιμές και 27 μη αρνητικές τιμές. Πιο συγκεκριμένα, αναπαρίστανται οι αρνητικές τιμές -1..-27 και οι μη αρνητικές τιμές 0..27-1. Προσέξτε πως η μικρότερη αρνητική τιμή, δηλαδή το -27, είναι κατ' απόλυτη τιμή μεγαλύτερη κατά μια μονάδα από τη μεγαλύτερη μη αρνητική, δηλαδή το 27-1. Αυτό συμβαίνει γιατί στις μη αρνητικές τιμές συμπεριλαμβάνεται και το 0.

Η Java για κάθε θεμελιώδη τύπο διαθέτει και μια κλάση (Class) που παρέχει λειτουργίες σχετικές με τον θεμελιώδη τύπο. Την έννοια της κλάσης θα αναλύσουμε με λεπτομέρεια κατά τη μελέτη του αντικειμενοστρεφούς μοντέλου. Για την ώρα, θα θεωρούμε την κλάση ως ένα σύνθετο τύπο ο οποίος, εκτός από δεδομένα, ενσωματώνει και λειτουργίες (συναρτήσεις). Σε γενικές γραμμές, θεμελιώδεις τύποι και αντίστοιχες κλάσεις έχουν την ίδια ονομασία με μόνη διαφορά ότι το όνομα της κλάσης ξεκινάει με κεφαλαίο ενώ του τύπου με πεζό. Έτσι ο τύπος byte συνοδεύεται από την κλάση Byte.

```
static void byteDemo() {
        System.out.println("Size of byte is " + Byte.SIZE + " bits");
        byte bMin = Byte.MIN_VALUE, bMax = Byte.MAX_VALUE, bLiteral = 123;
        System.out.println("bMin equals to " + bMin);
        System.out.println("bMax equals to " + bMax);
        System.out.println("bLiteral equals to " + bLiteral);
```

```
}
```

***Κώδικας 3.1**: Παράδειγμα χρήσης μεταβλητής τύπου byte*

Στον κώδικα 3.1, καταρχάς, εμφανίζουμε το μέγεθος του τύπου (Byte.SIZE) μετρημένο σε δυαδικά ψηφία (bits). Μετά δηλώνουμε τις μεταβλητές bMin, bMax και bLiteral. Την πρώτη αρχικοποιούμε στην ελάχιστη τιμή του τύπου (Byte.MIN_VALUE), τη δεύτερη στη μέγιστη τιμή του τύπου (Byte.MAX_VALUE) και την τρίτη σε μια ενδιάμεση τιμή. Την ελάχιστη και μέγιστη τιμή μας παρέχει η κλάση Byte που τις περιλαμβάνει ως σταθερές μεταβλητές (variable constants). Για την ενδιάμεση τιμή χρησιμοποιούμε μια κυριολεκτική σταθερά (literal). Στη συνέχεια στέλνουμε στη συσκευή εξόδου, δηλαδή στην τυπική περίπτωση στην οθόνη, το περιεχόμενο των bMin, bMax και bLiteral.
Παρατίθεται η έξοδος του κώδικα 3.1.

```
Size of byte is 8 bits
bMin equals to -128
bMax equals to 127
bLiteral equals to 123
```

short Ο τύπος short είναι προσημασμένος ακέραιος. Η κωδικοποίησή του είναι ανάλογη με αυτήν του τύπου byte, δηλαδή πρόσημο και μέτρο για τους θετικούς και συμπλήρωμα ως προς 2 για τους αρνητικούς. Το μήκος του όμως είναι 16 δυαδικών ψηφίων. Επομένως, οι τιμές που δέχονται οι μεταβλητές τύπου short είναι 0..215-1 και -1..-215.
Στον κώδικα 3.2, παρατίθεται παράδειγμα δήλωσης και αρχικοποίησης μεταβλητών τύπου short.

```
static void shortDemo() { //Κώδικας 3.2
        short s = Short.MIN_VALUE, s1 = Short.MAX_VALUE, s2 = 399;
        System.out.println("short s equals to " + s);
        System.out.println("short s1 equals to " + s1);
        System.out.println("short s2 equals to " + s2);
        System.out.println("Size of short is " + Short.SIZE + " bits");
}
```

***Κώδικας 3.2**: Παράδειγμα χρήσης μεταβλητών τύπου short*

Ακολουθεί η έξοδος του κώδικα 3.2

```
short s equals to -32768
short s1 equals to 32767
short s2 equals to 399
Size of short is 16 bits
```

int Ο τύπος int είναι προσημασμένος ακέραιος μήκους 32 δυαδικών ψηφίων. Η κωδικοποίησή του είναι ανάλογη με αυτήν του τύπου byte, δηλαδή πρόσημο και μέτρο για τους θετικούς και συμπλήρωμα ως προς 2 για τους αρνητικούς. Επομένως, οι τιμές που δέχονται οι μεταβλητές τύπου int είναι 0..231-1 και -1..-231.
Στον κώδικα 3.3, παρατίθεται παράδειγμα δήλωσης και αρχικοποίησης μεταβλητών τύπου int.

```
    static void intDemo() { //Κώδικας 3.3
        int iMin = Integer.MIN_VALUE,
                iMax = Integer.MAX_VALUE, iLiteral = 43127;
        System.out.println("int iMin equals to " + iMin);
        System.out.println("int iMax equals to " + iMax);
        System.out.println("int iLiteral equals to " + iLiteral);
        System.out.println("Size of int is " + Integer.SIZE + " bits");
    }
```

***Κώδικας 3.3**: Δήλωση και αρχικοποίηση μεταβλητών τύπου int*

Ακολουθεί η έξοδος του κώδικα 3.3.

```
int iMin equals to -2147483648
int iMax equals to 2147483647
int iLiteral equals to 43127
Size of int is 32 bits
```

Προσέξτε πως κατ' εξαίρεση, το όνομα της κλάσης που μας παρέχει πληροφορίες σχετικά με τον τύπο int, δεν είναι Int αλλά Integer.

long Ο τύπος long είναι προσημασμένος ακέραιος μήκους 64 δυαδικών ψηφίων. Η κωδικοποίησή του είναι ανάλογη με αυτήν του τύπου byte, δηλαδή πρόσημο και μέτρο για τους θετικούς και συμπλήρωμα ως προς 2 για τους αρνητικούς. Επομένως, οι τιμές που δέχονται οι μεταβλητές τύπου long είναι 0..263-1 και -1..-263.
Στον κώδικα 3.4, παρατίθεται παράδειγμα δήλωσης και αρχικοποίησης μεταβλητών τύπου long.

```
static void longDemo() { //Κώδικας 3.4
    long lMin = Long.MIN_VALUE,
            lMax = Long.MAX_VALUE, lLiteral = 2_147_483_699L;
    System.out.println("long lMin equals to " + lMin);
    System.out.println("long lMax equals to " + lMax);
    System.out.println("long lLiteral equals to " + lLiteral);
    System.out.println("Size of long is " + Long.SIZE + " bits");
}
```

***Κώδικας 3.4**: Δήλωση και αρχικοποίηση μεταβλητών τύπου long*

Ακολουθεί η έξοδος του κώδικα 3.4.

```
long lMin equals to -9223372036854775808
long lMax equals to 9223372036854775807
long lLiteral equals to 2147483699
Size of long is 64
```

Προσέξτε πως στην αρχικοποίηση της lLiteral, η κυριολεκτική σταθερά ακολουθείται από τον χαρακτήρα L. Οι ακέραιες κυριολεκτικές σταθερές που δεν ακολουθούνται από το L έχουν τύπο int. Αν θέλουμε να χρησιμοποιήσουμε κυριολεκτικές σταθερές με τύπο long, θα πρέπει αυτές να ακολουθούνται από το L. Στον κώδικα 3.4, αν αφαιρέσουμε το L, ο μεταγλωττιστής θα παράξει error, καθώς η τιμή 2147483699 είναι μεγαλύτερη από τη μέγιστη τιμή τύπου int. Ωστόσο σταθερές τύπου int (μέσα στα όρια του τύπου int) μπορούν να εκχωρηθούν σε μεταβλητές τύπου long, καθώς η Java κάνει αυτόματες μετατροπές μεταξύ συμβατών τύπων. Εναλλακτικά, αντί για κεφαλαίο L μπορεί να χρησιμοποιηθεί το πεζό l. Ωστόσο, συνίσταται η χρήση του κεφαλαίου καθώς το πεζό διακρίνεται με δυσκολία από τον αριθμό 1.

Προσέξτε επίσης πως ανάμεσα στα ψηφία έχουμε τοποθετήσει χαρακτήρες '_' (underscore). Από την έκδοση 7 και μετά, η Java επιτρέπει οποιονδήποτε αριθμό από underscores σε αριθμητικές σταθερές ώστε να διευκολύνεται η αναγνωσιμότητα των προγραμμάτων. Ωστόσο, δεν επιτρέπεται ο χαρακτήρας underscore, στην αρχή ή στο τέλος ενός αριθμού.

float Ο τύπος float χρησιμοποιείται για διαχείριση πραγματικών αριθμών. Έχει μήκος 32 δυαδικά ψηφία. Η κωδικοποίηση στη μνήμη των τιμών αυτού του τύπου ακολουθεί ένα πρότυπο γνωστό ως IEEE 754 και αναπτύχθηκε από το Ινστιτούτο Ηλεκτρολόγων και Ηλεκτρονικών Μηχανικών (Institute of Electrical and Electronics Engineers), που εδρεύει στη Νέα Υόρκη και στο οποίο συνήθως αναφερόμαστε με την αγγλική συντομογραφία IEEE. Η ανάλυση του προτύπου IEEE 754 είναι έξω από τα όρια αυτού του εγχειριδίου. Ωστόσο, είναι χρήσιμο να γνωρίζει κανείς πως οι πραγματικοί αριθμοί αναπαρίστανται σε εκθετική μορφή (Exponential Notation) και εκφράζονται με επιστημονικό συμβολισμό (scientific notation), mxEn, όπου m ένας πραγματικός αριθμός, n ακέραιος και E=10. Οι float αριθμοί χαρακτηρίζονται ως αριθμοί κινητής υποδιαστολής απλής ακρίβειας (single-precision floating point numbers).
Στον κώδικα 3.5, παρατίθεται παράδειγμα δήλωσης και αρχικοποίησης μεταβλητών τύπου float.

```
static void floatDemo() { //Κώδικας 3.5
```

```
        float fMin = Float.MIN_VALUE, fMax = Float.MAX_VALUE, fLit = 3.15f;
        System.out.println("float fMin equals to " + fMin);
        System.out.println("float fMax equals to " + fMax);
        System.out.println("float fLit equals to " + fLit);
        System.out.println("Size of float is " + Float.SIZE + " bits");
    }
```

Κώδικας 3.5*: Μεταβλητές τύπου float*

Ακολουθεί η έξοδος του κώδικα 3.5

```
float fMin equals to 1.4E-45
float fMax equals to 3.4028235E38
float fLit equals to 3.15
Size of float is 32 bits
```

Προσέξτε πως η σταθερά 3.15 ακολουθείται από το f, εναλλακτικά μπορεί να χρησιμοποιηθεί το F, ώστε να καθοριστεί ο τύπος της σταθεράς ως float. Χωρίς το f ή F, ο τύπος της σταθεράς είναι ο double που παρουσιάζεται αμέσως παρακάτω. Όπως βλέπουμε στην έξοδο του κώδικα 3.5, οι τιμές των fMin και fMax έχουν εκφραστεί με τον επιστημονικό συμβολισμό (scientific notation). Ο επιστημονικός συμβολισμός μπορεί να χρησιμοποιηθεί και για να εκφραστούν κυριολεκτικές σταθερές τύπου float ή double μέσα στον πηγαίο κώδικα. Επομένως, η ελάχιστη πραγματική τιμή που μπορεί να αναπαρασταθεί με τον float είναι 1.4E-45, δηλαδή 1.4x10-45 και η μέγιστη είναι 3.4028235E38, δηλαδή 3.4028235x1038, ο δε χώρος που καταλαμβάνεται στη μνήμη είναι 32 bits.

Οι μεταβλητές τύπου float δύνανται να λάβουν επιπλέον τις ειδικές τιμές Float.NEGATIVE_INFINITY, Float.POSITIVE_INFINITY και Float.NaN (Not a Number). Επιπλέον, σε αντίθεση με τους ακέραιους τύπους, οι πραγματικοί υποστηρίζουν δύο μηδέν, το +0f και το -0f. Αυτό οφείλεται στην κωδικοποίηση IEEE 754 που είναι συμμετρική μεταξύ θετικών και αρνητικών.

Για παράδειγμα, η τιμή Float.NEGATIVE_INFINITY επιστρέφεται όταν ένας αρνητικός αριθμός τύπου float διαιρείται με το +0f ή αντίστοιχα ένας θετικός float διαιρείται με το -0f. Επίσης, Float.NEGATIVE_INFINITY ή Float.POSITIVE_INFINITY μπορεί να επιστραφεί ως αποτέλεσμα έκφρασης στην οποία συμμετέχουν αυτές οι τιμές. Αν όμως έχουμε διαίρεση με Infinity, το αποτέλεσμα είναι NaN. Η τιμή Float.NaN επιστρέφεται και σε διάφορες άλλες περιπτώσεις, όπως για παράδειγμα όταν ζητηθεί η τετραγωνική ρίζα αρνητικού αριθμού.

Η διαίρεση ακεραίου με το 0 παράγει εξαίρεση[5] του τύπου “διαίρεση με το 0” (division by zero). Για παράδειγμα, ο κώδικας που ακολουθεί παράγει εξαίρεση.

```
int i = 1, j = 0;
System.out.println(i / j);
```

Αντίθετα, η διαίρεση πραγματικού με το 0 επιστρέφει Infinity. Για παράδειγμα, ο κώδικας που ακολουθεί τυπώνει Infinity.

```
float f=1, r=0;
System.out.println(f/r);
```

Αντίθετα, ο κώδικας

```
float nAN=(float)Math.sqrt(-1);
System.out.println(nAN);
```

θα τυπώσει NaN.

---

[5] Οι εξαιρέσεις είναι ο μηχανισμός διαχείρισης απροσδόκητων γεγονότων. Παρουσιάζονται αναλυτικά στην ενότητα 16. Για την ώρα θεωρήστε την εξαίρεση συνώνυμη του λάθους χρόνου εκτέλεσης.

Σημειώστε πως η συνάρτηση sqrt της κλάσης Math επιστρέφει την τετραγωνική ρίζα του ορίσματός της. Στη συγκεκριμένη κλήση επιστρέφει την τετραγωνική ρίζα του -1. Όπως είναι γνωστό, τέτοιος πραγματικός δεν υπάρχει, ως αποτέλεσμα η sqrt επιστρέφει NaN. Επίσης, ο τύπος της τιμής που επιστρέφει η sqrt είναι double. Επομένως, στην κλήση που συζητάμε θα επιστρέψει NaN τύπου double (Double.NaN). Για τον λόγο αυτόν αναγκαζόμαστε να προχωρήσουμε σε μετατροπή του τύπου από double σε float προκειμένου να εκχωρήσουμε την επιστρεφόμενη τιμή στη μεταβλητή nAN τύπου float. Όπως φαίνεται στον κώδικα, για να μετατρέψουμε το αποτέλεσμα της sqrt τοποθετήσαμε πριν την κλήση της το όνομα του νέου τύπου μέσα σε παρενθέσεις. Θα πρέπει να έχουμε υπόψη μας πως μια τέτοια μετατροπή γίνεται με ευθύνη του προγραμματιστή και μπορεί να οδηγήσει σε απώλεια ακρίβειας.

Η κωδικοποίηση IEEE 754 μας δίνει τη δυνατότητα στο ίδιο μήκος μνήμης να αναπαραστήσουμε μεγαλύτερους αριθμούς από ότι η κωδικοποίηση συμπληρώματος ως προς 2 που χρησιμοποιείται για την κωδικοποίηση των ακέραιων. Ωστόσο, είναι σημαντικό να κατανοήσουμε ότι οι πραγματικοί αριθμοί αποθηκεύονται στη μνήμη προσεγγιστικά. Ως αποτέλεσμα συγκρίσεις πραγματικών με χρήση του τελεστή ισότητας (equality operator), ==, ή των υπόλοιπων σχεσιακών τελεστών (>, <, >=, <= και !=, ενότητα 4.1.4) δεν είναι αξιόπιστες και πρέπει να αποφεύγονται.
Για παράδειγμα, ο κώδικας

```
float f=0.1f+0.000000001f;
System.out.println(f==0.1f);
```

θα τυπώσει true.
Προσέξτε τον ακόλουθο κώδικα

```
float d1 = 1, d2=1;
for (int j = 1; j <= 10; j++) {
     d1 = d1+0.00001f;
}
d2=d2+0.00001f * (10);
System.out.println(d1==d2);
```

Οι μεταβλητές d1 και d2 αρχικοποιούνται και οι δύο στην τιμή 1. Σε κάθε βήμα της επαναληπτικής διαδικασίας for, προστίθεται η τιμή 0.00001f στην d1. Έχουμε ένα σύνολο 10 επαναλήψεων, άρα στην d1 προστίθεται 10 φορές το 0.00001. Στη συνέχεια προσθέτουμε στην d2 το 10*0.00001. Αναμένουμε επομένως ότι τα d1 και d2 είναι ίσα μεταξύ τους. Ωστόσο, ο κώδικας θα τυπώσει false, καθώς η τιμή της d1 έχει υπολογιστεί ως 1.0001001 και της d2 ως 1.0001. Για αυτόν τον λόγο, ανάλογα με τις ανάγκες της εφαρμογής μας, η σύγκριση μεταξύ πραγματικών αριθμών μπορεί να γίνεται προσεγγιστικά.

```
static boolean approximateEquals(float f1, float f2, float
epsilon) {
     return Math.abs(f2 - f1) < epsilon;
}
```

Χρησιμοποιήστε τη συνάρτηση approximateEquals για να συγκρίνετε αριθμούς float. Για να δείτε το αποτέλεσμά της, καλέστε την ως εξής:

```
System.out.println(approximateEquals(d1, d2, 0.0001));
```

Η τρίτη παράμετρος συνιστά μια οριακή τιμή. Αν η απόλυτη τιμή της διαφοράς των δύο float είναι μικρότερη από αυτήν την οριακή τιμή, οι αριθμοί θεωρούνται ίσοι. Αν για παράδειγμα θέλουμε να συγκρίνουμε τα μήκη δύο οδών και το μήκος της πρώτης το μετρήσουμε στα 2 km και αυτό της δεύτερης στα 2 km και 1 χιλιοστό του εκατοστού, μπορεί να τις θεωρήσουμε ισομήκεις.

Γενικότερα, ο τύπος float θα πρέπει να αποφεύγεται για διαχείριση τιμών που απαιτούν ακρίβεια. Για τέτοιου τύπου πληροφορίες συνίσταται η χρήση της κλάσης Java.math.BigDecimal. Το ίδιο ισχύει και για τον τύπο double που παρουσιάζεται αμέσως παρακάτω.

double Πρόκειται για τύπο αποθήκευσης πραγματικών αριθμών, επίσης κωδικοποιημένο με το IEEE 754. Η διαφορά του από τον float είναι πως ο double έχει μήκος 64 bits. Πρόκειται για τύπο κινητής υποδιαστολής διπλής ακρίβειας (double-precision 64-bit IEEE 754 floating point). Εξαιτίας του μεγαλύτερου μεγέθους του μπορεί να φιλοξενήσει πραγματικές τιμές με μεγαλύτερη ακρίβεια από τον float. Στην τυπική περίπτωση είναι ο τύπος που θα πρέπει να χρησιμοποιούμε όταν θέλουμε να διαχειριστούμε πραγματικούς αριθμούς. Ο float θα πρέπει να χρησιμοποιείται μόνο στις περιπτώσεις που έχουμε πολλά δεδομένα από πραγματικούς αριθμούς, χρειάζεται να κάνουμε οικονομία στη μνήμη και είμαστε σίγουροι πως η ακρίβεια που προσφέρει, μας αρκεί. Ο τύπος double διαθέτει επίσης τιμές Double.PositiveInfinity, Double.NegativeInfinity, Double.NaN, αρνητικό και θετικό μηδέν με παρόμοια χαρακτηριστικά με τις αντίστοιχες τιμές του float.
Στον κώδικα 3.6 παρατίθεται παράδειγμα δήλωσης και αρχικοποίησης μεταβλητών τύπου double.

```
    static void doubleDemo() { //Κώδικας 3.6
        double dMin = Double.MIN_VALUE, dMax = Double.MAX_VALUE, dLit =
3.4028235E38;
        System.out.println("double dMin equals to " + dMin);
        System.out.println("double dMax equals to " + dMax);
        System.out.println("double dLit equals to " + dLit);
        System.out.println("Size of double is " + Double.SIZE + "bits");
    }
```

***Κώδικας 3.6**: Δήλωση και αρχικοποίηση μεταβλητών τύπου double*

Ακολουθεί η έξοδος του κώδικα 3.6.

```
double dMin equals to 4.9E-324
double dMax equals to 1.7976931348623157E308
double dLit equals to 3.4028235E38
Size of double is 64bits
```

Επομένως, η ελάχιστη τιμή του double είναι 4.9x10-324, η μέγιστη 1.7976931348623157x10308 και το μέγεθός του 64 bits. Παρατηρήστε πως στην αρχικοποίηση της dLit έχει χρησιμοποιηθεί επιστημονικός συμβολισμός. Οι σταθερές με δεκαδικά που δεν έχουν χαρακτηριστεί ως float θεωρούνται εξ' ορισμού πως είναι τύπου double. Μπορεί να συνοδευτεί μια τέτοια σταθερά από τον χαρακτήρα D ή d προκειμένου να δηλωθεί ο τύπος της σαφώς, ωστόσο αυτό αποφεύγεται από σύμβαση.

Παρότι ο double προσφέρει μεγαλύτερη ακρίβεια από τον float, δεν παύει να αντιπροσωπεύει προσεγγιστικά τους πραγματικούς αριθμούς και η χρήση του θα πρέπει να αποφεύγεται όπου μας ενδιαφέρει η ακρίβεια και στη θέση του να χρησιμοποιείται η κλάση Java.math.BigDecimal. Για προσεγγιστική ισότητα χρησιμοποιήστε τη συνάρτηση

```
static boolean approximateEquals(double d1, double d2, double
epsilon) {
     return Math.abs(d2 - d1) < epsilon;
}
```

Ίσως να προσέξατε πως οι συναρτήσεις MIN_VALUE επιστρέφουν αρνητικές τιμές για τους ακέραιους τύπους και θετικές τιμές για τους float και double. Πράγματι, για τους πραγματικούς οι συναρτήσεις αυτές επιστρέφουν την πλησιέστερη τιμή στο 0. Αν χρειάζεστε την ελάχιστη αρνητική τιμή ενός πραγματικού τύπου, χρησιμοποιήστε –Float.MAX_VALUE και –Double.MAX_Value, αντίστοιχα.

boolean Οι μεταβλητές του τύπου boolean λαμβάνουν αυστηρά 2 δυνατές τιμές, true ή false. Ένα δυαδικό ψηφίο αρκεί για την αποθήκευση των τιμών αυτού του τύπου. Ωστόσο, οι προδιαγραφές της Java δεν καθορίζουν το μέγεθος του τύπου. Επομένως, το μέγεθος του τύπου boolean εξαρτάται από τη συγκεκριμένη υλοποίηση. Σε κάποιες γλώσσες είναι δυνατή η μετατροπή μεταξύ boolean και ακέραιων τύπων, στην Java δεν είναι επιτρεπτή τέτοια μετατροπή τύπου [1].
Στον κώδικα 3.7, παρουσιάζουμε παράδειγμα δήλωσης και αρχικοποίησης μεταβλητών τύπου boolean:

```
    static void booleans() { //Κώδικας 3.7
```

```
        boolean b1 = true, b2 = false;
        System.out.println("b1 equals to " + b1);
        System.out.println("b2 equals to " + b2);
    }
```

***Κώδικας 3.7**: Δήλωση και αρχικοποίηση boolean μεταβλητών*

Ακολουθεί η έξοδος του κώδικα 3.7.

```
b1 equals to true
b2 equals to false
```

char Ο τύπος char χρησιμοποιείται για την αποθήκευση χαρακτήρων. Υποστηρίζει το σύνολο Unicode, δηλαδή τους χαρακτήρες από τις περισσότερες γλώσσες συν μια ποικιλία χαρακτήρων που δεν συνιστούν γράμματα αλφάβητου και διάφορους χαρακτήρες ελέγχου (control characters) που είναι συνήθως μη εκτυπώσιμοι, όπως για παράδειγμα, ο χαρακτήρας που σηματοδοτεί το τέλος της τρέχουσας γραμμής (End Of Line). Στον κώδικα 3.8 δίνουμε παράδειγμα χρήσης μεταβλητών του τύπου char.

```
1 static void charDemo() { //Κώδικας 3.8
2         char cMin = Character.MIN_VALUE, cMax = Character.MAX_VALUE, cLit = 'A',
c = '\u0043';
3         System.out.println(cMin + " " + (int) cMin);
4         System.out.println(cMax + " " + (int) cMax);
5         System.out.println(c + " " + (int) c);
6         System.out.println(cLit + " " + (int) cLit);
7         System.out.println((char) (cLit + 1));
8     }
```

***Κώδικας 3.8**: Παράδειγμα χρήσης του τύπου char*

Ο κώδικας 3.8, έχει την ακόλουθη έξοδο

```
 0
□ 65535
C 67
A 65
B
```

Προσέξτε πως η κλάση που αντιστοιχεί στον τύπο char ονομάζεται Character. Σε κάθε μια από τις γραμμές 2-5 τυπώνουμε τον χαρακτήρα και στη συνέχεια μια τιμή που προκύπτει από τη μετατροπή του χαρακτήρα σε int. Η μετατροπή αυτή είναι ορθή γιατί οι χαρακτήρες κωδικοποιούνται στη μνήμη ως 16-bit μη προσημασμένοι ακέραιοι. Επομένως, ο char είναι και αυτός ένας ακέραιος τύπος. Πιο συγκεκριμένα, μια μεταβλητή τύπου char κρατάει την ακέραιη τιμή που συνιστά τον κωδικό του χαρακτήρα στο Unicode. Η έξοδος της γραμμής 2 φαίνεται ως ένα διάστημα ακολουθούμενο από το 0. Αυτό συμβαίνει γιατί ο cMin είναι ο NULL χαρακτήρας ο οποίος, όταν στέλνεται σε συσκευή εξόδου δεν κάνει τίποτα. Επομένως τυπώνεται το διάστημα που στέλνουμε στη γραμμή 2 και μετά ο κωδικός Unicode του NULL, δηλαδή το 0. Η γραμμή 3 τυπώνει ένα τετραγωνάκι ακολουθούμενο από τον κωδικό 65535 που στο δεκαεξαδικό σύστημα είναι το FFFF. Σημειώστε πως οι θέσεις του Unicode από FFF0 έως και FFFF χρησιμοποιούνται για ειδικές περιπτώσεις [3] που δεν ενδιαφέρουν στα πλαίσια αυτού του εγχειριδίου.

Προσέξτε πως η μεταβλητή c έχει αρχικοποιηθεί με τη σταθερά ‘\u0043’. Είναι εναλλακτικός τρόπος για έκφραση σταθερών τύπου char. Σύμφωνα με τον αυτόν τον τρόπο, ένας χαρακτήρας μπορεί να αναπαρασταθεί με \u ακολουθούμενο από τον δεκαεξαδικό κωδικό του. Θα δούμε περισσότερα σε σχέση με αυτό, αμέσως παρακάτω, στους χαρακτήρες διαφυγής. Η γραμμή 5 τυπώνει τον χαρακτήρα ‘A’ με τον κωδικό του. Στη γραμμή 6, προσθέτουμε το cLit που περιέχει το A με το 1. Η πρόσθεση αυτή είναι δυνατή, γιατί όπως εξηγήσαμε, στη μνήμη ένας char διατηρεί μια ακέραια τιμή που στη συγκεκριμένη περίπτωση είναι η τιμή 65. Επομένως, η τιμή της έκφραση cLit+1 είναι ίση με 66. Όταν το 66 μετατρέπεται σε char, στην ίδια γραμμή, τότε το αποτέλεσμα είναι ο χαρακτήρας B.

3.3 Ακολουθίες ελέγχου (Control Sequences)

Ο χαρακτήρας \ (backslash) ακολουθούμενος από έναν ειδικό χαρακτήρα ονομάζεται ακολουθία ελέγχου (control sequence) ή χαρακτήρας διαφυγής (escape character).
Στον πίνακα 3.1 παρουσιάζονται οι ακολουθίες ελέγχου της Java.

\t	tab	Αφήνει ένα διάστημα
\b	backspace	Μεταφέρει την επόμενη έξοδο έναν χαρακτήρα πίσω
\n	newline	Μεταφέρει την επόμενη έξοδο στην επόμενη γραμμή
\r	carriage return	Μεταφέρει την επόμενη έξοδο στην αρχή της τρέχουσας γραμμής
\f	form feed	Αλλαγή σελίδας στον εκτυπωτή
\'	single quote	Ο ειδικός χαρακτήρας ', λαμβάνεται ως κανονικός
\"	double quote	Ο ειδικός χαρακτήρας ", λαμβάνεται ως κανονικός
\\	backslash	Ο ειδικός χαρακτήρας \, λαμβάνεται ως κανονικός
\u	unicode character	Ακολουθούμενο από αριθμητική σταθερά αναπαριστά τον αντίστοιχο Unicode χαρακτήρα

Πίνακας 3.1 *Ακολουθίες Ελέγχου*

Ακολουθεί παράδειγμα χρήσης των ακολουθιών ελέγχου:

```
9     static void controlSequences() { //Κώδικας 3.9
10         System.out.println("slash_t" + '\t' + "second ");
11         System.out.println("slash_b" + '\b' + "second ");
12         System.out.println("slash_n" + '\n' + "second ");
13         System.out.println("slash_r" + '\r' + "second ");
14         System.out.println("slash_singlequote" + '\'' + "second ");
15         System.out.println("slash_doublequote" + '\"' + "second ");
16         System.out.println("slash_backslash" + '\\' + "second ");
17         System.out.println("slash_unicode" + '\u0041' + "second ");
18     }
```

Κώδικας 3.9*: Παραδείγματα χρήσης ακολουθιών ελέγχου*

Η έξοδος του κώδικα έχει ως εξής:

```
slash_t         second
slash_second
slash_n
second
second
slash_singlequote'second
slash_doublequote"second
slash_backslash\second
slash_unicodeAsecond
```

Η γραμμή 10 του κώδικα 3.9, τυπώνει “slash_t”, μετά αφήνει ένα tab και τέλος τυπώνει τη σειρά “second”.
Στη γραμμή 11, ο χαρακτήρας ‘\b’ ΄φέρνει τη σειρά “second” έναν χαρακτήρα πιο πίσω με αποτέλεσμα την εξαφάνιση του b από την “slash_b”.
Στη γραμμή 12, ο χαρακτήρας newline έχει ως αποτέλεσμα να εκτυπωθεί η “slash_n” και η “second” να εκτυπωθεί στην επόμενη γραμμή.
Στη γραμμή 13, το carriage return φέρνει την έξοδο στην αρχή της γραμμής με αποτέλεσμα να εξαφανίζεται η σειρά “slash_r”.
Η γραμμή 14 τυπώνει ένα quote ανάμεσα στην “slash_singlequote” και στην “second”.
Η γραμμή 15 τυπώνει ένα double quote ανάμεσα στην “slash_doublequote” και στην “second”.
Η γραμμή 16 τυπώνει ένα backslash ανάμεσα στην “slash_backslash” και στην “second”.
Η γραμμή 17 τυπώνει τον χαρακτήρα ‘A’ ανάμεσα στην “slash_unicode” και στην “second”.

3.4 Αναγνωριστικά

Είναι προφανές από την μέχρι τώρα παρουσίαση πως κάθε τοπική μεταβλητή λαμβάνει ένα όνομα που αποτελεί το αναγνωριστικό (identifier) της. Η σύνθεση των αναγνωριστικών υπακούει σε κανόνες. Πιο συγκεκριμένα, ο πρώτος χαρακτήρας υποχρεωτικά πρέπει να είναι ένας χαρακτήρας του Αγγλικού αλφάβητου, ο χαρακτήρας $ (dollar) ή ο χαρακτήρας _ (underscore). Στη συνέχεια, επιπλέον των προαναφερόμενων χαρακτήρων, μπορούν να χρησιμοποιηθούν και ψηφία.
Στον πίνακα 3.2, δίνονται παραδείγματα από έγκυρα και μη έγκυρα αναγνωριστικά

Έγκυρα Αναγνωριστικά	**Μη έγκυρα**
rVal	!color
costPerUnit	1start
event23	*profit
net_weight	start-up
$score	last/digit

Πίνακας 3.2 *Παραδείγματα έγκυρων και άκυρων αναγνωριστικών*

Εκτός από αυτούς τους κανόνες που είναι υποχρεωτικοί καθώς επιβάλλονται από τον μεταγλωττιστή, υπάρχει και μια σειρά από συμβάσεις που διέπουν την ονοματολογία των αναγνωριστικών των μεταβλητών. Οι συμβάσεις δεν επιβάλλονται από τον μεταγλωττιστή, έχουν ωστόσο καθολική αποδοχή και θα πρέπει να ακολουθούνται συστηματικά.

Από σύμβαση, τα αναγνωριστικά των μεταβλητών πρέπει να αρχίζουν με πεζό χαρακτήρα του αγγλικού αλφάβητου. Επομένως, οι χαρακτήρες dollar και underscore πρέπει να αποφεύγονται από την πρώτη θέση. Ειδικότερα ο χαρακτήρας dollar πρέπει να αποφεύγεται και από τις υπόλοιπες θέσεις. Θα πρέπει να χρησιμοποιούνται πλήρεις λέξεις ώστε ο κώδικας να είναι αναγνώσιμος και αυτοτεκμηριωμένος (self-documented). Για παράδειγμα, το αναγνωριστικό totalCost είναι πολύ σαφέστερο σε σχέση με τη σύντμηση tC. Η Java διαφοροποιεί μεταξύ πεζών και κεφαλαίων (case sensitive). Το αναγνωριστικό totalCost είναι διαφορετικό από το αναγνωριστικό totalcost. Αν ένα αναγνωριστικό μεταβλητής αποτελείται από μια λέξη, τότε όλα τα γράμματα στην ονομασία του πρέπει να είναι πεζοί χαρακτήρες. Αν όμως αποτελείται από περισσότερες από μία λέξεις, όλες οι επόμενες πρέπει να αρχίζουν από κεφαλαίο, π.χ. counter, customerWeight, first, firstOccurance.

Οι κανόνες και οι συμβάσεις για τα αναγνωριστικά των μεταβλητών ισχύουν και για τις παραμέτρους. Αντίθετα, όλοι οι χαρακτήρες στα αναγνωριστικά των σταθερών μεταβλητών πρέπει να είναι κεφαλαίοι.

3.5 Υπερχείλιση

Κατά την επιλογή ενός αριθμητικού τύπου, θα πρέπει να είμαστε σίγουροι πως η χωρητικότητά του επαρκεί για τα δεδομένα που θέλουμε να διαχειριστούμε. Σε περίπτωση που επιχειρήσουμε να εκχωρήσουμε τιμή έξω από τα όρια του τύπου, το αποτέλεσμα θα είναι να υπερχειλίσει (overflow) η μνήμη της μεταβλητής.
Στον κώδικα 3.10, παρουσιάζουμε ένα παράδειγμα υπερχείλισης:

```
static void overflow() { //Κώδικας 3.10
    int i = Integer.MAX_VALUE;
    i = i + 1;
    System.out.println(i);
    System.out.println(Integer.MIN_VALUE - 1);
}
```

Κώδικας 3.10*: Υπερχείλιση ακέραιων*

Η έξοδος της overflow είναι

```
-2147483648
2147483647
```

Η έξοδος αυτή είναι το αποτέλεσμα της υπερχείλισης. Καθώς τα αθροίσματα υπολογίζονται κατά τον χρόνο εκτέλεσης, ο μεταγλωττιστής δεν είναι σε θέση να μας προειδοποιήσει. Ως αποτέλεσμα έχουμε την εισαγωγή ενός λάθους χρόνου εκτέλεσης (run-time error) στο πρόγραμμά μας. Τα λάθη χρόνου εκτέλεσης είναι πολύ πιο σοβαρά από τα λάθη μεταγλώττισης ακριβώς γιατί δεν εντοπίζονται κατά τη μεταγλώττιση και ταξιδεύουν ως τον χρήστη του λογισμικού μας όπου μπορεί να προξενήσουν σημαντική ζημιά. Φανταστείτε τις πιθανές επιπτώσεις ενός τέτοιου λάθους σε ένα λογισμικό πλοήγησης αεροσκάφους.

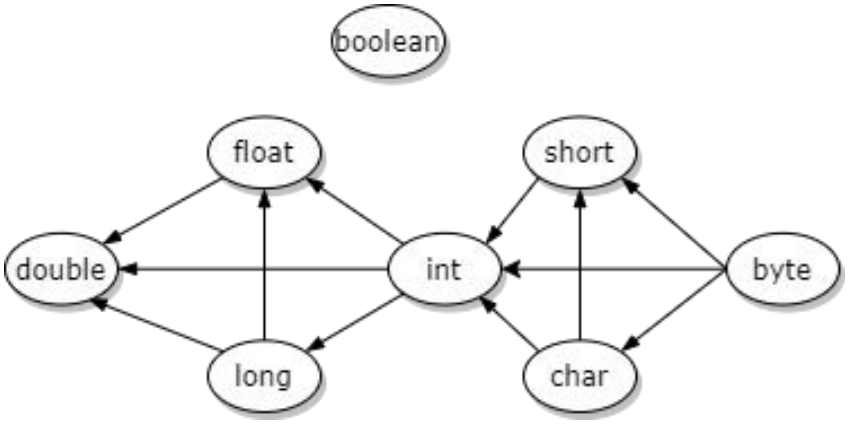
Αν παρόλα αυτά θέλετε να προσθέσετε δύο int και δεν είστε βέβαιοι ότι το άθροισμά τους είναι στα όρια του τύπου, μπορείτε να κάνετε μετατροπή τύπου τουλάχιστον στον ένα από τους δύο και στη συνέχεια να εκχωρήσετε το αποτέλεσμα σε έναν long. Στον κώδικα 3.11 δίνεται σχετικό παράδειγμα:

```
static void notOverflow() { //Κώδικας 3.11
    int i = Integer.MAX_VALUE;
    long k = (long) i + 1;
    System.out.println(k);
}
```

***Κώδικας 3.11**: Άθροισμα ακέραιων εκτός ορίων του τύπου*

Η μετατροπή του τύπου του ενός ορίσματος σε long έχει ως αποτέλεσμα ο τύπος του αθροίσματος να είναι long. Επομένως, αν επιχειρήσουμε να εκχωρήσουμε το αποτέλεσμα αυτής της έκφρασης σε int, ο μεταγλωττιστής θα μας αποτρέψει. Αν όμως, όπως στο παράδειγμα, εκχωρήσουμε το άθροισμα σε μεταβλητή τύπου long, τότε θα λάβουμε το σωστό αποτέλεσμα, δηλαδή στο παράδειγμα τον αριθμό 2147483648.

Στην Java, ο τύπος των εκφράσεων στις οποίες συμμετέχουν οι αριθμητικοί τύποι byte και short είναι int. Επομένως, όταν προσθέτουμε 2 μεταβλητές τύπου short ή δύο τύπου byte ή μια short με μια byte, δεν χρειάζεται να μετατρέψουμε κανέναν τύπο, καθώς το αποτέλεσμα είναι int. Αυτό σημαίνει πως το αποτέλεσμα δεν μπορεί παρά να εκχωρηθεί σε μεταβλητή συμβατού τύπου, μεγαλύτερης ή ίσης χωρητικότητας από τον int. Μπορούμε ωστόσο να εκχωρήσουμε μια μεταβλητή ενός τύπου σε μεταβλητή άλλου τύπου με την προϋπόθεση πως οι τύποι είναι συμβατοί μεταξύ τους και ο τύπος της μεταβλητής που τοποθετείται δεξιά από τον τελεστή εκχώρησης, =, έχει μικρότερη χωρητικότητα από τον τύπο της μεταβλητής που βρίσκεται αριστερά. Σε αυτές τις περιπτώσεις, η Java προχωράει σε αυτόματη μετατροπή τύπου (automatic type conversion). Στο σχήμα 2, απεικονίζονται οι αυτόματες μετατροπές μεταξύ των θεμελιωδών τύπων.



***Σχήμα 3.2**: Αυτόματες μετατροπές μεταξύ θεμελιωδών τύπων*

Όπως φαίνεται στο σχήμα 3.2, σε μια μεταβλητή τύπου double μπορεί να εκχωρηθεί τιμή οποιουδήποτε αριθμητικού τύπου, σε float οποιουδήποτε πλην του double, σε long οποιουδήποτε εκτός από double και float, σε int μπορούν να εκχωρηθούν short, char και byte και φυσικά int, σε short μπορούν να εκχωρηθούν char, byte και short, σε char μπορούν να εκχωρηθούν byte και char και σε byte μόνο byte. Ο τύπος boolean δεν εκχωρείται παρά μόνο σε boolean και κανένας άλλος τύπος δεν μετατρέπεται σε boolean. Περισσότερες πληροφορίες για τις μετατροπές τύπων μπορείτε να βρείτε στις προδιαγραφές της Java [4], §5.1.2, 5.1.3, 5.1.4.

Η υπερχείλιση ισχύει και για τους πραγματικούς τύπους, float και double. Ωστόσο, η συμπεριφορά εδώ είναι διαφορετική. Για παράδειγμα, η έκφραση Double.MAX_VALUE+1d επιστρέφει Double.MAX_VALUE. Μάλιστα, το ίδιο αποτέλεσμα έχουμε αν προσθέσουμε στον Double.MAX_VALUE

και μεγαλύτερες τιμές. Μόνο αν προσθέσουμε τιμή μεγαλύτερη ή ίση του 2970, το αποτέλεσμα της έκφρασης θα γίνει Infinity. Επίσης, η έκφραση Double.MIN_VALUE-1 επιστρέφει -1. Ο Double.MIN_VALUE είναι ένας πάρα πολύ μικρός αριθμός, τόσο μικρός ώστε μπορούμε να θεωρήσουμε ότι προσεγγίζει το μηδέν. Επομένως, αν από το 0 αφαιρέσουμε το 1 θα λάβουμε -1. Αντίστοιχα, αν διαιρέσουμε το Double.MIN_VALUE με το 2 θα λάβουμε ως αποτέλεσμα 0. Η περίεργη εκ πρώτης όψεως συμπεριφορά των πραγματικών μεταβλητών οφείλεται στον τρόπο αναπαράστασής τους. Προσέξτε πως μεταξύ δύο διαδοχικών πραγματικών αριθμών, x1 και x2, που μπορούν να αναπαρασταθούν με την κωδικοποίηση της IEEE, μεσολαβεί άπειρο πλήθος πραγματικών τιμών. Όλες αυτές οι τιμές κωδικοποιούνται σαν x1 ή σαν x2. Ο προσεγγιστικός χαρακτήρας των πραγματικών αριθμών δεν οφείλεται ειδικά στην κωδικοποίηση IEEE. Ένα διαφορετικό σύστημα κωδικοποίησης θα μπορούσε να παρέχει μεγαλύτερη ακρίβεια αλλά ο προσεγγιστικός χαρακτήρας παραμένει. Εξάλλου μεταξύ 2 οποιωνδήποτε πραγματικών αριθμών, μεσολαβεί πάντα άπειρο πλήθος πραγματικών τιμών.

3.6 Αριθμητικές σταθερές

Για την έκφραση των κυριολεκτικών αριθμητικών σταθερών μπορούν να χρησιμοποιηθούν, πέραν του δεκαδικού συστήματος και το δυαδικό, το οκταδικό και δεκαεξαδικό σύστημα αρίθμησης. Όταν μια αριθμητική σταθερά αρχίζει από 0b και ακολουθείται από δυαδικά ψηφία, τότε ερμηνεύεται στο δυαδικό σύστημα. Ο κώδικας

```
int i = 0b10, j = 0b11;
System.out.println(i + j);
```

έχει ως αποτέλεσμα να εκτυπωθεί 5, καθώς προσθέτει το 10 με βάση το 2 που είναι ίσο με το 2 στο δεκαδικό σύστημα, με το 11 του δυαδικού που είναι ίσο με το 3 του δεκαδικού.
Όταν μια αριθμητική σταθερά αρχίζει από 0 και ακολουθείται από τα ψηφία του οκταδικού συστήματος, ερμηνεύεται σαν οκταδικός αριθμός. Ο κώδικας

```
int i=010, j=011;
System.out.println(i+j);
```

έχει ως αποτέλεσμα να εκτυπωθεί 17, καθώς προσθέτει το 10 με βάση το 8 που είναι ίσο με το 8 στο δεκαδικό σύστημα με το 11 του οκταδικού που είναι ίσο με το 9 του δεκαδικού.
Όταν μια αριθμητική σταθερά αρχίζει από 0x και ακολουθείται από τα ψηφία του δεκαεξαδικού συστήματος, ερμηνεύεται σαν δεκαεξαδικός αριθμός. Ο κώδικας

```
int i=0x10, j=0x11;
System.out.println(i+j);
```

έχει ως αποτέλεσμα να εκτυπωθεί 33, καθώς προσθέτει το 10 με βάση το 16 που είναι ίσο με το 16 στο δεκαδικό σύστημα, με το 11 του δεκαεξαδικού που είναι ίσο με το 17 του δεκαδικού.

3.7 Αλφαριθμητικές Σειρές

Σε πολλές περιπτώσεις χρειάζεται να διαχειριστούμε δεδομένα όπως επωνυμίες και μηνύματα, δηλαδή σειρές από αλφαριθμητικούς χαρακτήρες. Για αυτές τις περιπτώσεις, η Java παρέχει τους τύπους String, StringBuffer και StringBuilder.

Το String δεν είναι θεμελιώδης τύπος στην Java, λόγω όμως της εκτεταμένης χρήσης του και της ανάγκης να αξιοποιείται σε παραδείγματα από τα πρώτα βήματα στη μελέτη της γλώσσας, παρουσιάζονται εδώ κάποιες αναγκαίες πληροφορίες. Ήδη στο πρώτο μας πρόγραμμα στην ενότητα 3, χρησιμοποιήσαμε σταθερές τύπου String. Ο κώδικας

```
System.out.println("Hello World!");
```

χρησιμοποιεί την αλφαριθμητική σταθερά "Hello World!" για να τυπώσει το κατάλληλο μήνυμα στην οθόνη.

Οι σταθερές τύπου String εσωκλείονται σε διπλά εισαγωγικά (double quotes), σε αντίθεση με τις σταθερές τύπου char που εσωκλείονται σε μονά εισαγωγικά. Μια πράξη που γίνεται πολύ συχνά ανάμεσα σε String είναι η σύνδεση των String (concatenation). Η σύνδεση των String γίνεται με τη βοήθεια του τελεστή +.

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

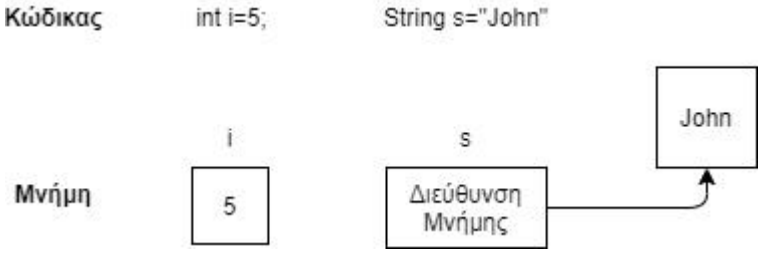
Κώδικας 3.11*: Παράδειγμα μεταβλητών τύπου String*

Στον κώδικα 3.11, στην πρώτη γραμμή, δηλώνουμε και αρχικοποιούμε τη μεταβλητή s1 τύπου String, στη δεύτερη γραμμή, δηλώνουμε και αρχικοποιούμε τη μεταβλητή s2. Τέλος, στην τρίτη γραμμή, συνδέουμε το s1 με μια σταθερά τύπου String που αποτελείται μόνο από τον χαρακτήρα διάστημα (space) και το αποτέλεσμα αυτής της σύνδεσης το συνδέουμε με το String s2. Ως αποτέλεσμα, τυπώνεται Hello World.

Προσέξτε πως η ονομασία του τύπου String ξεκινά με κεφαλαίο χαρακτήρα. Αυτό οφείλεται στο ότι ο τύπος δεν είναι θεμελιώδης, αλλά συνιστά μια κλάση και όπως έχουμε επισημάνει, από σύμβαση τα ονόματα των κλάσεων ξεκινούν με κεφαλαίο.

3.8 Μεταβλητές τιμής και αναφοράς

Μια άλλη πολύ ουσιαστική διαφορά μεταξύ String και θεμελιωδών τύπων είναι πως οι μεταβλητές τύπου String είναι μεταβλητές αναφοράς (reference variables) σε αντίθεση με τις μεταβλητές των θεμελιωδών τύπων που είναι μεταβλητές τιμής (value variables). Όταν σε μία μεταβλητή π.χ. τύπου int εκχωρούμε μια τιμή, τότε στη μνήμη της μεταβλητής εκχωρείται αυτή καθαυτή η τιμή. Αντίθετα, όταν σε μεταβλητή τύπου String εκχωρούμε μια τιμή, τότε δεσμεύεται χώρος στη μνήμη δυναμικά, στον χώρο αυτόν αποθηκεύεται η τιμή ενώ στη μεταβλητή αποθηκεύεται η διεύθυνση της τιμής, όπως δείχνει το σχήμα 3.3.



Σχήμα 3.3 *Μεταβλητές τιμής και αναφοράς*

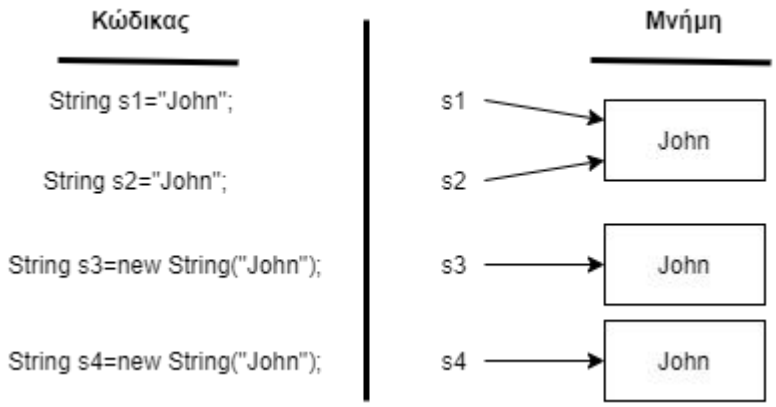
Στην τυπική περίπτωση, οι μεταβλητές αναφοράς δημιουργούνται με τη χρήση της λέξης κλειδί new. Για παράδειγμα, η δημιουργία ενός String γίνεται ως εξής:

```
String s=new String("John");
```

Ειδικά για τον τύπο String και για τις κλάσεις των θεμελιωδών τύπων παρέχεται η δυνατότητα δημιουργίας μεταβλητών χωρίς χρήση της λέξης new. Για παράδειγμα,

```
String s="John";
Integer i=125;
```

Όμως οι δύο τρόποι έχουν ουσιαστική διαφορά. Κάθε φορά που χρησιμοποιείται η new για τη δημιουργία μιας μεταβλητής δεσμεύεται νέος χώρος για την καταχώριση της τιμής της μεταβλητής. Όταν όμως η δημιουργία των μεταβλητών γίνεται χωρίς το new, τότε η ίδια τιμή αποθηκεύεται στην ίδια διεύθυνση μνήμης.



Σχήμα 3.4 *Δημιουργία μεταβλητών με την new*

Όπως δείχνει το σχήμα 3.4, οι μεταβλητές s1 και s2 δημιουργήθηκαν χωρίς χρήση της λέξης κλειδί new. Εφόσον έχουν την ίδια τιμή, κατανέμεται ο ίδιος χώρος στη μνήμη. Αντίθετα, οι μεταβλητές s3 και s4 δημιουργήθηκαν με χρήση της new. Σε αυτήν την περίπτωση κατανέμεται διαφορετικός χώρος στη μνήμη ακόμη και αν τα περιεχόμενα των μεταβλητών είναι ίδια. Επομένως, η s1 και η s2 περιέχουν την ίδια διεύθυνση μνήμης και άρα ισχύει s1==s2, δηλαδή οι μεταβλητές συγκρινόμενες με τον τελεστή ισότητας (ενότητα 4.1.4.1) θεωρούνται ίσες. Οι μεταβλητές s3 και s4 αναφέρονται στο ίδιο περιεχόμενο που είναι κατανεμημένο όμως σε διαφορετικές θέσεις στη μνήμη. Επομένως, στην s3 αποθηκεύεται διαφορετική διεύθυνση από ότι στην s4 και συνεπώς δεν ισχύει η ισότητα μεταξύ s3 και s4.
Αυτή η διαχείριση έχει ως στόχο τη βελτίωση της αποτελεσματικότητας του κώδικα και βασίζεται στο γεγονός ότι οι τιμές του τύπου String και των κλάσεων των θεμελιωδών τύπων είναι αμετάβλητες (immutable), δηλαδή από τη στιγμή που θα δημιουργηθεί μια τιμή ενός τέτοιου τύπου διασφαλίζεται πως δεν θα μεταβληθεί ως το τέλος του προγράμματος. Οι αμετάβλητοι τύποι εξυπηρετούν στον πολυνηματικό προγραμματισμό και χαρακτηρίζονται ως νηματικά ασφαλείς (thread safe).

3.8.1 Συλλέκτης απορριμμάτων

Εδώ όμως προκύπτει ένα άλλο θέμα. Το θέμα της απελευθέρωσης της μνήμης από τιμές μεταβλητών αναφοράς που δεν είναι χρήσιμες πλέον. Οι τοπικές μεταβλητές όπως είδαμε καταστρέφονται στο τέλος του μπλοκ στο οποίο δηλώθηκαν. Αυτό ισχύει είτε είναι μεταβλητές τιμής είτε μεταβλητές αναφοράς. Τι γίνεται όμως με τις τιμές των μεταβλητών αναφοράς που δεσμεύουν χώρο στη μνήμη δυναμικά; Είναι ένα θέμα που στην τυπική περίπτωση δεν απασχολεί ιδιαίτερα τον προγραμματιστή της Java καθώς η διαχείριση της δυναμικής μνήμης στην Java γίνεται από τον συλλέκτη απορριμμάτων (garbage collector). Ο συλλέκτης απορριμμάτων θεωρεί τις τιμές αναφοράς ως επιλέξιμες για αποδέσμευση μνήμης όταν δεν υπάρχει προς αυτές καμιά αναφορά μέσα στο πρόγραμμα.

```
static void garbageCollection() { //Κώδικας 3.4
    String s = new String("John");
    s = new String("Mary");
    System.out.println(s);
}
```

Κώδικας 3.4 Επιλέξιμες τιμές αναφοράς για αποδέσμευση μνήμης

Στην τελευταία γραμμή της garbageCollection στον κώδικα 3.4, το String John είναι ήδη επιλέξιμο προς αποδέσμευση. Η μόνη αναφορά στο John είναι η s έτσι όπως δημιουργείται στην πρώτη γραμμή. Στη δεύτερη γραμμή όμως, η αναφορά s ανακατευθύνεται σε άλλη τιμή. Έτσι το String John μένει χωρίς καμία αναφορά, δηλαδή είναι αδύνατο να το προσπελάσουμε μέσα από το πρόγραμμά μας καθώς το πρόγραμμά μας δεν γνωρίζει τη διεύθυνσή του. Με το τέλος της εκτέλεσης της garbageCollection, η τοπική μεταβλητή s καταστρέφεται, οπότε μένει και το String Mary χωρίς αναφορά και άρα καθίσταται και αυτό επιλέξιμο προς αποδέσμευση.

Το πότε ακριβώς θα πραγματοποιηθεί η αποδέσμευση της μνήμης εξαρτάται από τον συλλέκτη απορριμμάτων ο οποίος έχει δικούς του αλγόριθμους με στόχο την αποδοτική διαχείριση της μνήμης.

3.9 Ασκήσεις

1. Περνάει με επιτυχία μεταγλώττιση η x1;

```
static void x1() {
     int k = 1, K = 2;
     System.out.println(k + " " + K);
}
```

Παραβιάζει κάποια από τις συμβάσεις ονοματολογίας; Εξηγήστε την απάντησή σας.

2. Περνάει με επιτυχία μεταγλώττιση η x2; Εξηγήστε την απάντησή σας.

```
static void x2() {
     int k = 1, j;
     j = j + 1;
     System.out.println(k + " " + j);
}
```

3. Περνάει με επιτυχία μεταγλώττιση η x3; Εξηγήστε την απάντησή σας.

```
static void x3() {
     int k = 1;
     final int j = k;
     j = j + 1;
     System.out.println(k + " " + j);
}
```

4. Περνάει με επιτυχία μεταγλώττιση η x4; Εξηγήστε την απάντησή σας.

```
static void x4() {
     int k = 1;
     {
          k = 2;
     }
     System.out.println(k);
}
```

5. Περνάει με επιτυχία μεταγλώττιση η x5; Εξηγήστε την απάντησή σας.

```
static void x5() {
     int k = 1;
     {
          int k = 2;
     }
     System.out.println(k);
}
```

6. Ποια είναι η έξοδος της x6;

```
static void x6() {
     int k = 010;
     System.out.println(k);
}
```

7. Ποια είναι η έξοδος της x7;

```
static void x7() {
     int k = 010, j = 10;
     System.out.println(k + j);
}
```

8. Ποια είναι η έξοδος της x8;

```
static void x8() {
     int k = 010, j = 0x10;
     System.out.println(k + j);
}
```

9. Ποια από τα παρακάτω ονόματα μεταβλητών παραβιάζουν τους κανόνες και ποια τις συμβάσεις ονοματολογίας;

```
int sum; char Start; final char END; boolean ExitLoop;
int finalsum; double 3oAthroisma; byte day; float float_10;
float _1_;
```

10. Υπάρχει διαφορά μεταξύ των κυριολεκτικών σταθερών 8 και 8.0;

Υποδείξεις: Προσπαθήστε να απαντήσετε χωρίς τη βοήθεια του μεταγλωττιστή. Χρησιμοποιήστε τον μεταγλωττιστή για να ελέγξετε τις απαντήσεις σας.

Βιβλιογραφία

[1] “Primitive Data Types (The Java™ Tutorials > Learning the Java Language > Language Basics).” https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (accessed Sep. 14, 2021).

[2] “Java All-in-One For Dummies, 6th Edition | Wiley,” Wiley.com. https://www.wiley.com/en-bb/Java+All+in+One+For+Dummies%2C+6th+Edition-p-9781119680451 (accessed Sep. 14, 2021).

[3] “Unicode 13.0.0.” https://unicode.org/versions/Unicode13.0.0/ (accessed Sep. 14, 2021).

[4] “The Java® Language Specification, Chapter 5. Conversions and Contexts, §5.1.2, §5.1.3, §5.1.4.” https://docs.oracle.com/javase/specs/jls/se10/html/index.html (accessed Sep. 14, 2021).

Κεφάλαιο 4

Σύνοψη
Σε αυτήν την ενότητα παρουσιάζονται οι τελεστές της Java και αναλύεται η συμπεριφορά τους. Πιο συγκεκριμένα, αναλύονται ο τελεστής εκχώρησης (assignment), οι αριθμητικοί τελεστές (arithmetic operators), ο τελεστής σύνδεσης συμβολοσειρών (concatenation), οι μοναδιαίοι (unary) τελεστές, ο τελεστής ισότητας (equality), οι σχεσιακοί (relational) τελεστές, ο τριαδικός τελεστής συνθήκης (ternary), ο τελεστής ελέγχου τύπου (instanceof) καθώς και οι τελεστές ψηφίου προς ψηφίο (Bitwise) και κύλισης ψηφίου (shift operators). Επιπλέον, παρουσιάζονται βασικοί τρόποι διαμόρφωσης της εξόδου στην οθόνη και εισαγωγής δεδομένων από το πληκτρολόγιο.

Προαπαιτούμενη γνώση
Προαπαιτείται να έχετε εγκατεστημένο το NetBeans με Java έκδοσης μεγαλύτερης ή ίσης της 8 και να γνωρίζετε τους θεμελιώδεις τύπους της Java, να χρησιμοποιείτε τον τύπο String, τις τοπικές μεταβλητές και σταθερές.

Λέξεις κλειδιά
Τελεστής, Τελεστέος, Εκχώρηση.

4 Τελεστές και βασική Είσοδος και Έξοδος

Η Java ενσωματώνει μια σειρά από συναρτήσεις που χρησιμεύουν ώστε να μπορούμε να κάνουμε διάφορες πράξεις (operations). Κάθε πράξη περιλαμβάνει τον τελεστή (operator) και τους τελεστέους (operands). Για παράδειγμα, στην πράξη 5+3, τελεστής είναι το σύμβολο της πρόσθεσης, + και τελεστέοι τα 5 και 3. Προκειμένου να διευκολυνθεί η έκφραση στο πρόγραμμα τέτοιων πράξεων, η Java παρέχει ειδικό συντακτικό κλήσης των τελεστών έτσι ώστε οι ανάλογες εκφράσεις να μην ακολουθούν το τυπικό συντακτικό κλήσης συναρτήσεων αλλά να διατυπώνονται κατά τρόπο που ομοιάζει με την εκτεταμένη χρήση τους στα μαθηματικά και άλλους τομείς.

Ένα άλλο ζήτημα, αναγκαίο για τα προγράμματά μας, είναι η βασική είσοδος και έξοδος. Πρόκειται για τις διαδικασίες που επιτρέπουν την επικοινωνία και αλληλεπίδραση μεταξύ χρήστη και προγράμματος. Πώς δηλαδή θα πάρω δεδομένα από τον χρήστη και πώς θα παρουσιάσω σε αυτόν τις πληροφορίες που θέλω;

Σε αυτό το κεφάλαιο παρουσιάζουμε τους βασικούς τελεστές που υποστηρίζει η Java [1] ταξινομώντας τους σε κατηγορίες ανάλογα με τα χαρακτηριστικά τους καθώς και τις βασικές λειτουργίες εισόδου από το πληκτρολόγιο και εξόδου στην οθόνη.

4.1 Τελεστές

Οι τελεστές (operators) στην Java οργανώνονται ανάλογα με τον αριθμό τελεστέων (operands) που δέχονται. Έτσι, έχουμε τους μοναδιαίους (unary) τελεστές, δηλαδή αυτούς που δέχονται έναν τελεστέο, τους δυαδικούς (binary), δηλαδή αυτούς με δύο τελεστέους και έναν τριαδικό (ternary) τελεστή. Επίσης, οι τελεστές κατηγοριοποιούνται σε αριθμητικούς (arithmetic), υποθετικούς (conditional) ή λογικούς (logical), σχεσιακούς (relational), τελεστές ψηφίου (bitwise) και τους τελεστές εκχώρησης (assignment).
Σε μια αλγεβρική έκφραση είναι δυνατό να συνδυάζονται περισσότεροι από έναν τελεστές. Για παράδειγμα, στην αλγεβρική έκφραση 2+3x5, συνδυάζονται οι τελεστές πρόσθεσης και πολλαπλασιασμού. Όπως γνωρίζουμε από την άλγεβρα, ο πολλαπλασιασμός έχει προτεραιότητα σε σχέση με την πρόσθεση. Επομένως, 2+3x5=17. Παρόμοια και στις εκφράσεις (expressions) της Java είναι δυνατό να συνδυάζονται πολλοί τελεστές. Επομένως, για τον συνεπή υπολογισμό των εκφράσεων έχουν καθοριστεί προτεραιότητες και για τους τελεστές της Java. Στον πίνακα 4.1 παρουσιάζονται οι τελεστές διατεταγμένοι από υψηλότερη σε χαμηλότερη προτεραιότητα. Τελεστές στην ίδια γραμμή του πίνακα 4.1 έχουν ίση προτεραιότητα.

Τύπος	**Λειτουργία**	**Προτεραιότητα**
Μοναδιαίος	Μεταθεματική αύξηση/μείωση	++, --
	Προθεματική αύξηση/μείωση, συμπλήρωμα, λογικό NOT	++, --, ~, !

Αριθμητικοί	Πολλαπλασιασμός	*, /, %
	Πρόσθεση	+, -
Διολίσθησης	Διολίσθηση	<<, >>, >>>
Σχεσιακοί	Σύγκριση	<, >, <=, >=, instanceof
	Έλεγχος ισότητας	==, !=
Ψηφίου	AND ψηφίο προς ψηφίο	&
	XOR ψηφίο προς ψηφίο	^
	OR ψηφίο προς ψηφίο	|
Λογικοί	Λογικό AND	&&
	Λογικό OR	||
Τριαδικός		? :
Εκχώρησης	Εκχώρησης	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

***Πίνακας 4.1**: Προτεραιότητα τελεστών*

Ας υποθέσουμε την αλγεβρική έκφραση x=5+3-2. Η πρόσθεση και η αφαίρεση έχουν ίδια προτεραιότητα μεταξύ τους και ψηλότερη από την εκχώρηση. Επομένως, πρώτα θα υπολογιστεί η έκφραση 5+3-2 και το αποτέλεσμά της θα εκχωρηθεί στη μεταβλητή x. Πώς όμως θα αξιολογηθεί το τμήμα 5+3-2; Ο κανόνας εδώ είναι πως η αξιολόγηση γίνεται από αριστερά προς τα δεξιά, δηλαδή γίνεται πρώτα η πρόσθεση και το αποτέλεσμά της συνιστά τελεστέο για την αφαίρεση. Το ίδιο ισχύει για τις περισσότερες εκφράσεις της Java. Οι υπολογισμοί γίνονται από αριστερά προς τα δεξιά για όλους τους τελεστές πλην της εκχώρησης όπου η αξιολόγηση γίνεται από δεξιά προς τα αριστερά.

4.1.1 Ο τελεστής εκχώρησης

Η εκχώρηση είναι δυαδική πράξη και ο βασικός τελεστής της συμβολίζεται με τον χαρακτήρα '='. Στη γραμμή 2 του κώδικα 4.1 δηλώνονται τρεις ακέραιες μεταβλητές και αρχικοποιούνται οι δύο από αυτές. Στη γραμμή 3, εκχωρείται η τιμή 8 στη μεταβλητή i. Συνηθίζεται η πρώτη εκχώρηση σε μεταβλητή να αποκαλείται αρχικοποίηση (initialization). Η έκφραση (expression) στη γραμμή 3, επομένως, μπορούμε να πούμε πως αρχικοποιεί την i. Η αξιολόγηση γίνεται από τη δεξιά προς την αριστερή πλευρά της έκφρασης. Αυτό σημαίνει πως πρώτα θα υπολογιστεί ότι υπάρχει στα δεξιά του τελεστή εκχώρησης και στη συνέχεια η τιμή που θα προκύψει θα εκχωρηθεί στην i. Βέβαια, στη συγκεκριμένη περίπτωση, δεν απαιτούνται ιδιαίτεροι υπολογισμοί δεδομένου ότι στα δεξιά του τελεστή υπάρχει μια αριθμητική σταθερά. Θα μπορούσε όμως να υπάρχει και μια έκφραση που το αποτέλεσμα της αξιολόγησής της θα ήταν ακέραιος. Στη συνέχεια αυτής της ενότητας θα δούμε αρκετά τέτοια παραδείγματα.

```
1     static void assignmentDemo() { //Κώδικας 4.1
2         int i, j = 5, k = 9;
3         i = 8;
4         System.out.println(j = i);
5         j = k = i;
6         System.out.println(i + " " + j + " " + k);
7     }
```

***Κώδικας 4.1**: Ο τελεστής εκχώρησης*

Εκείνο όμως που έχει ιδιαίτερο ενδιαφέρον είναι πως η εκχώρηση επιστρέφει στο περιβάλλον κλήσης της μια τιμή και συγκεκριμένα την τιμή της i μετά την εκχώρηση. Αυτή η τιμή δεν αξιοποιείται στη γραμμή 3. Αξιοποιείται όμως στη γραμμή 4 που τυπώνει την έκφραση j=i. Αν τρέξετε τον κώδικα 4.1, θα διαπιστώσετε πως η τιμή που τυπώνεται είναι 8. Η εκχώρηση, αφού αξιολόγησε την τιμή του i, την εκχώρησε στην j και επέστρεψε στην println την τιμή του j μετά την εκχώρηση. Σε αυτό το χαρακτηριστικό βασίζεται και η υποστήριξη της πολλαπλής εκχώρησης (γραμμή 5).

Όπως φαίνεται στον πίνακα 4.1, είναι διαθέσιμοι και άλλοι τελεστές εκχώρησης. Η παρουσίασή τους γίνεται πιο κάτω σε αυτήν την ενότητα.

4.1.2 Αριθμητικοί τελεστές

Η Java παρέχει τους δυαδικούς τελεστές πρόσθεση, αφαίρεση, πολλαπλασιασμό, διαίρεση, υπόλοιπο διαίρεσης. Όλοι αυτοί οι τελεστές είναι δυαδικοί.

4.1.2.1 Πρόσθεση

Ο τελεστής πρόσθεσης συμβολίζεται με τον χαρακτήρα '+'. Στον κώδικα 4.2, προσθέτουμε το i με το j. Το αποτέλεσμα στέλνεται στην έξοδο όπου και τυπώνεται η τιμή 5. Στη γραμμή 20 όμως, το αποτέλεσμα ίσως φανεί λίγο παράδοξο. Πράγματι, η έξοδος του κώδικα της γραμμής 4 είναι i+j=32. Ο χαρακτήρας '+' χρησιμοποιείται τόσο για την πρόσθεση μεταξύ αριθμών, όσο και για τη σύνδεση String (concatenation) μεταξύ τους. Το αν θα ληφθεί ως πρόσθεση αριθμών ή ως σύνδεση String εξαρτάται από τους τελεστέους που συντάσσονται μαζί του. Αν έστω και ένας από τους τελεστέους είναι String, τότε εκτελείται η σύνδεση Strings. Στη γραμμή 4, η αξιολόγηση γίνεται από αριστερά προς τα δεξιά. Έτσι, στην έκφραση "i+j="+i εντοπίζεται πρώτα το String "i+j=", οπότε λαμβάνεται η απόφαση πως πρόκειται για σύνδεση. Έτσι ο επόμενος ακέραιος τελεστέος μετατρέπεται αυτόματα σε String, έτσι το αποτέλεσμα "i+j="+i είναι "i+j=3" και ο τύπος του είναι String. Στη συνέχεια μένει να συνδεθεί αυτό το String με την ακέραιη μεταβλητή j. Με παρόμοια διαδικασία, το τελικό αποτέλεσμα είναι "i+j=32".

```
1     static void additionDemo() { //Κώδικας 4.2
2         int i = 3, j = 2;
3         System.out.println(i + j);
4         System.out.println("i+j= " + i + j); //concatenation
5         System.out.println("i+j= " + (i + j));
6         String h = "hello", w = "World";
7         System.out.println(h + " " + w);
8         double k = 1;
9         float m = 1f;
10        k = k + m;
11        char a = 'A';
12        System.out.println(a + 1);
13        System.out.println((char) (a + 1));
14    }
```

***Κώδικας 4.2**: Ο τελεστής πρόσθεσης*

Αν εκείνο που θέλουμε να πετύχουμε είναι να προστεθούν πρώτα το i με το j και το αποτέλεσμα να συνδεθεί με το "i+j= " ώστε να εκτυπωθούν μαζί, τότε θα πρέπει να δώσουμε προτεραιότητα στην πρόσθεση των ακέραιων, όπως δείχνει η γραμμή 5. Ισχύει και εδώ, όπως και στην άλγεβρα πως οι εκφράσεις μέσα σε παρενθέσεις αξιολογούνται κατά προτεραιότητα. Στη συνέχεια, οι γραμμές 22 και 23 παρουσιάζουν μια περίπτωση σύνδεσης String. Οι γραμμές, 8, 9 και 10 επιδεικνύουν πως ο τελεστής πρόσθεσης συντάσσεται με όλους τους αριθμητικούς τύπους. Ωστόσο, θα πρέπει να έχουμε υπόψη πως όταν συντάσσεται με δύο διαφορετικούς αριθμητικούς τύπους, ο τύπος του αποτελέσματος αποκτάται από τον τελεστέο που έχει το μεγαλύτερο μέγεθος. Στο παράδειγμά μας, η έκφραση k+m επιστρέφει double. Αν επιχειρούσαμε να εκχωρήσουμε αυτό το αποτέλεσμα σε μεταβλητή τύπου float, ο μεταγλωττιστής θα παρήγαγε λάθος. Όπως έχουμε αναφέρει στην ενότητα 3.2, το αποτέλεσμα των εκφράσεων με byte ή short είναι αυτόματα int.
Ο τελεστής της πρόσθεσης δέχεται και τελεστέους τύπου char. Στην περίπτωση αυτή προστίθεται ο κωδικός των χαρακτήρων και το αποτέλεσμα της πράξης είναι τύπου int. Έτσι η γραμμή 12 θα προσθέσει στο 65 που είναι ο κωδικός του 'A' το 1 και θα τυπώσει το 66 που είναι ο κωδικός του 'B'. Μπορούμε να δούμε τον χαρακτήρα 'B' αν μετατρέψουμε τον τύπο του αποτελέσματος (type casting) όπως δείχνει η γραμμή 13.

4.1.2.2 Αφαίρεση

Ο τελεστής αφαίρεσης συμβολίζεται με τον χαρακτήρα '-'. Η γραμμή 3 του κώδικα 4.3 παράγει λάθος μεταγλώττισης εφόσον αποσχολιαστεί. Πράγματι, ανάλογα με όσα είπαμε στην περιγραφή του τελεστή πρόσθεσης, η έκφραση "i-j="+i επιστρέφει String. Στη συνέχεια η αφαίρεση ενός ακεραίου από ένα String δεν ορίζεται. Επομένως, σε αυτήν την περίπτωση, είμαστε αναγκασμένοι να δώσουμε προτεραιότητα στην αφαίρεση όπως φαίνεται στη γραμμή 4. Η αφαίρεση συντάσσεται και με δεδομένα τύπου char. Η

συμπεριφορά της σε αυτήν την περίπτωση είναι ανάλογη με την πρόσθεση. Επίσης, ανάλογη με την πρόσθεση συμπεριφορά παρουσιάζει η αφαίρεση στην περίπτωση τελεστέων διαφορετικών αριθμητικών τύπων.

```
1    static void subtractionDemo() { //Κώδικας 4.3
2        int i = 3, j = 2;
3        //System.out.println("i-j= "+i-j);
4        System.out.println("i-j= " + (i - j));
5        char bita = 'B';
6        System.out.println(bita - 1);
7    }
```

***Κώδικας 4.3**: Ο τελεστής αφαίρεσης*

4.1.2.3 Πολλαπλασιασμός

Ο τελεστής του πολλαπλασιασμού συμβολίζεται με τον χαρακτήρα αστερίσκο, '*'. Το αποτέλεσμα της γραμμής 4 του κώδικα 4.4 είναι το αναμενόμενο. Αυτό οφείλεται στο γεγονός ότι ο πολλαπλασιασμός έχει προτεραιότητα έναντι της πρόσθεσης. Επομένως, στη γραμμή 4 υπολογίζεται πρώτα το γινόμενο i*j και στη συνέχεια συνδέεται το String "i*j=" με το αποτέλεσμα του πολλαπλασιασμού. Κατά συνέπεια οι παρενθέσεις που περικλείουν το γινόμενο i*j, στη γραμμή 5, είναι περιττές χωρίς να οδηγούν σε λάθος. Ο πολλαπλασιασμός δέχεται και τελεστέους τύπου char. Το αποτέλεσμα είναι το γινόμενο των κωδικών των χαρακτήρων. Έτσι, η γραμμή 6 θα τυπώσει 132, δηλαδή τον κωδικό του B επί 2.

```
1    static void multiplicationDemo() { //Κώδικας 4.4
2       int i = 3, j = 2;
3       char bita = 'B';
4       System.out.println("i*j= " + i * j);
5       System.out.println("i*j= " + (i * j));
6       System.out.println("bita*2=" + bita * 2);
7    }
```

***Κώδικας 4.4**: Ο τελεστής του πολλαπλασιασμού*

4.1.2.4 Διαίρεση

Ο τελεστής της διαίρεσης συμβολίζεται με τον χαρακτήρα πλάγια κάθετος, '/'. Ό,τι αναφέρθηκε για τον πολλαπλασιασμό, ισχύει κατ' αντιστοιχία και για τη διαίρεση. Επιπλέον, η διαίρεση διακρίνεται σε διαίρεση μεταξύ πραγματικών και διαίρεση μεταξύ ακέραιων. Αν οι τελεστέοι είναι ακέραιοι, τότε έχουμε ακέραια διαίρεση. Αν ένας τουλάχιστον από τους τελεστέους είναι πραγματικός, τότε έχουμε διαίρεση.

```
1     static void divisionDemo() { //Κώδικας 4.5
2        int i = 3;
3        int j = 2;
4        System.out.println("i/j= " + i / j);
5        System.out.println("i/j= " + (i / j));
6        System.out.println();
7        double d = 3, b = 2;
8        System.out.println("d/b= " + (d / b));
9        System.out.println("(double)i/j= " + (double) i / j);
10    }
```

***Κώδικας 4.5**: Ο τελεστής της διαίρεσης*

Η έξοδος της γραμμής 4 του κώδικα 4.5 είναι i/j=1. Και οι δύο τελεστέοι είναι ακέραιοι, οπότε η Java εκτελεί διαίρεση ακέραιων. Αντίθετα στη γραμμή 8, η έξοδος είναι d/b=1.5. Αν επιθυμούμε να κάνουμε διαίρεση πραγματικών μεταξύ ακέραιων μεταβλητών, θα πρέπει να μετατρέψουμε τον τύπο τουλάχιστον σε έναν από τους τελεστέους όπως φαίνεται στη γραμμή 9.

4.1.2.4 Υπόλοιπο διαίρεσης

Το υπόλοιπο συμβολίζεται με τον χαρακτήρα επί τοις εκατό, '%'. Εδώ έχουμε διαφοροποίηση μεταξύ υπολοίπου που προκύπτει από διαίρεση πραγματικών και υπολοίπου διαίρεσης ακέραιων.

```
1     static void moduloDemo() { //Κώδικας 4.6
2        int i = 3, j = 2; //1
3        System.out.println("i%j= " + i % j);
4        System.out.println("i%j= " + (i % j));
5        double d = 7.5, b = 3.5;
6        System.out.println("d%b= " + d % b);
7        System.out.println("d%b= " + (d % b));
8     }
```

***Κώδικας 4.6**: Το υπόλοιπο διαίρεσης*

Η έξοδος της γραμμής 3 του κώδικα 4.6 είναι i%j=1. Ο τύπος της έκφρασης i%j είναι int. Στη γραμμή 6 εκτελείται διαίρεση μεταξύ πραγματικών και η έξοδος είναι d%b=0.5. Ο τύπος της έκφρασης d%b είναι double.

4.1.3 Μοναδιαίοι τελεστές

Οι μοναδιαίοι τελεστές που υποστηρίζονται είναι το θετικό πρόσημο, το αρνητικό πρόσημο, οι τελεστές προσαύξησης, οι τελεστές μείωσης και ο λογικός τελεστής αντιστροφής.

4.1.3.1 Οι τελεστές προσήμου

Ο μοναδιαίος τελεστής μείον (unary minus operator) συμβολίζεται με τον χαρακτήρα '-' και επιστρέφει τον αντίθετο του τελεστέου. Στη γραμμή 73, ο κώδικας 4.7, τυπώνει -1. Ο τελεστής μείον δέχεται ως τελεστέους και πραγματικούς αλλά και χαρακτήρες. Ανάλογη είναι η λειτουργία του μοναδιαίου τελεστή συν (unary plus operator), '+'. Ωστόσο, ο τελεστής συν, ισχύει εξ ορισμού και έτσι δεν αναφέρεται σχεδόν ποτέ σε κώδικες Java.

```
1     static void unaryOps() { //Κώδικας 4.7
2        int i = 1;
3        System.out.println(i);
4        System.out.println(+i);
5        System.out.println(-i);
6        System.out.println();
7
8        System.out.println("Postfix increment");
9        System.out.println(i++);
10       System.out.println(i);
1        System.out.println();
12
13        System.out.println("Prefix increment");
14        i = 1;
15        System.out.println(++i);
16        System.out.println(i);
17        System.out.println();
18
19        System.out.println("Postfix decrement");
20        i = 1;
21        System.out.println(i--);
22        System.out.println(i);
23        System.out.println();
24
25        System.out.println("Prefix decrement");
26        i = 1;
27        System.out.println(--i);
28        System.out.println(i);
29        System.out.println();
```

```
30
31          boolean b = false;
32          System.out.println("b= " + b + ", !b= " + !b);
33      }
```

Κώδικας 4.7*: Μοναδιαίοι τελεστές*

4.1.3.2 Τελεστές προσαύξησης

Ο τελεστής προσαύξησης συμβολίζεται με τους χαρακτήρες "++" και παρέχεται σε δύο μορφές, στον προθεματικό τελεστή προσαύξησης (prefix increment operator) και στον μεταθεματικό τελεστή προσαύξησης (postfix increment operator). Όταν ο τελεστής προηγείται του τελεστέου, έχουμε τον προθεματικό τελεστή. Αντίθετα, έχουμε τον μεταθεματικό όταν ο τελεστέος προηγείται του τελεστή. Η λειτουργία των δύο τελεστών διαφέρει. Ο προθεματικός τελεστής αυξάνει κατά 1 την τιμή του τελεστέου και επιστρέφει την αυξημένη τιμή. Αντίθετα, ο μεταθεματικός τελεστής αυξάνει την τιμή του τελεστέου κατά 1, επιστρέφει όμως την τιμή που είχε ο τελεστέος πριν την αύξηση. Ο κώδικας στη γραμμή 9 θα τυπώσει 1. Ο μεταθεματικός τελεστής αύξησε την τιμή του i κατά 1 αλλά επέστρεψε την τιμή που είχε το i πριν την αύξηση. Στη συνέχεια στη γραμμή 10 θα τυπωθεί η αυξημένη τιμή, δηλαδή το 2. Αντίθετα, στη γραμμή 15 θα τυπωθεί 2. Επίσης, 2 θα τυπωθεί στη γραμμή 16.

4.1.3.3 Τελεστές μείωσης

Παρόμοια είναι η λειτουργία των τελεστών μείωσης. Ο μεταθεματικός τελεστής μείωσης μειώνει την τιμή του τελεστέου κατά 1, επιστρέφει όμως την τιμή πριν τη μείωση. Αντίθετα, ο προθεματικός τελεστής μείωσης μειώνει την τιμή του τελεστέου κατά 1 και επιστρέφει την τιμή μετά τη μείωση.
Στη γραμμή 21 του κώδικα 4.7, θα τυπωθεί 1 ενώ στη γραμμή 22 θα τυπωθεί 0. Αντίθετα, στις γραμμές 27 και 28 θα τυπωθεί 0.

4.1.3.4 Ο λογικός τελεστής αντιστροφής

Ο λογικός τελεστής αντιστροφής (logical complement operator) συμβολίζεται με τον χαρακτήρα '!' και συντάσσεται με τύπο boolean. Το αποτέλεσμα του τελεστή είναι η αντιστροφή της τιμής του τελεστέου. Έτσι, ο κώδικας στη γραμμή 32, θα τυπώσει b= false, !b= true.

4.1.4 Σχεσιακοί τελεστές

Υποστηρίζονται οι ακόλουθοι σχεσιακοί τελεστές: ο τελεστής ελέγχου ισότητας, ο τελεστής ελέγχου ανισότητας, ο τελεστής μεγαλύτερο από, ο τελεστής μεγαλύτερο ή ίσο, ο τελεστής μικρότερο από και ο τελεστής μικρότερο ή ίσο. Όλοι οι σχεσιακοί τελεστές είναι δυαδικοί και επιστρέφουν boolean τιμή.

4.1.4.1 Ο τελεστής ελέγχου ισότητας

Ο τελεστής ελέγχου ισότητας (equality operator) συμβολίζεται με "==" και επιστρέφει true αν και μόνο αν οι τελεστέοι του έχουν την ίδια τιμή. Στην περίπτωση που οι τελεστέοι είναι ακέραιοι τύποι ή boolean, η λειτουργία του τελεστή είναι απλή. Η γραμμή 3 του κώδικα 4.8, θα τυπώσει i==j is true.

Στην περίπτωση όμως πραγματικών τελεστέων δεν ισχύει το ίδιο. Στη γραμμή 4, ορίζουμε τρεις μεταβλητές τύπου double. Η μεταβλητή d αρχικοποιείται στην τιμή 1+2*k. Στη συνέχεια, στις γραμμές 107 και 108 προσθέτουμε στην b την k, δηλαδή προσθέτουμε δύο φορές το k. Σύμφωνα με τα μαθηματικά αναμένουμε πως το d είναι ίσο με το b. Ωστόσο, λόγω του προσεγγιστικού χαρακτήρα και της διαφορετικής σειράς που γίνονται οι πράξεις κατά τον υπολογισμό των d και b, το d δεν είναι ίσο με το b. Η έξοδος της γραμμής 7 είναι b=1.0000200000000001 and d=1.00002. Λογική συνέπεια είναι πως η γραμμή 8 τυπώνει false. Ο γενικός κανόνας είναι πως η χρήση του τελεστή ισότητας με πραγματικούς τελεστέους πρέπει να αποφεύγεται. Αντίθετα, για τη σύγκριση πραγματικών αριθμών θα πρέπει να χρησιμοποιείται κατάλληλη συνάρτηση προσεγγιστικής ισότητας που παρουσιάστηκε στην ενότητα 3.2.

Η συμπεριφορά του τελεστή με τύπους αναφοράς φαίνεται διαφορετική καθώς στην περίπτωση αυτή συγκρίνονται διευθύνσεις και όχι περιεχόμενα. Οι αναφορές τύπου String s1 και s2 δείχνουν στο ίδιο String στη μνήμη το περιεχόμενο του οποίου είναι "John". Επομένως η γραμμή 11 τυπώνει s1==s2 is true. Αντίθετα,

η s3 αναφέρεται σε ένα String, που, παρότι το περιεχόμενό του είναι “John”, βρίσκεται σε άλλη θέση της μνήμης. Επομένως, η γραμμή 12 τυπώνει s1==s3 is false. Αν θέλουμε να συγκρίνουμε για ισότητα δύο Strings ως προς το περιεχόμενό τους, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση equals όπως φαίνεται στη γραμμή 13. Η έξοδος της γραμμής 13 είναι s1.equals(s3) is true. Κατά αντίστοιχο τρόπο λειτουργεί ο τελεστής ισότητας για όλους τους μη θεμελιώδεις τύπους. Κάθε μεταβλητή αναφοράς που έχει λάβει την τιμή της από την επιστροφή της new δεν μπορεί να συγκριθεί με τον τελεστή ισότητας αλλά με τη συνάρτηση equals.

```
1     static void equalityOp() { //Κώδικας 4.8
2         int i = 1, j = 1;
3         System.out.println("i==j is " + (i == j));
4         double k = 0.00001, d = 1 + 2 * k, b = 1;
5         b = b + k;
6         b = b + k;
7         System.out.println("b=" + b + " and d=" + d);
8         System.out.println(b == d);
9         System.out.println(Double.compare(b,d));
10        String s1 = "John", s2 = "John", s3 = new String("John");
11        System.out.println("s1==s2 is " + s1 == s2);
12        System.out.println("s1==s3 is " + s1 == s3);
13        System.out.println("s1.equals(s3) is " + s1.equals(s3));
14    }
```

***Κώδικας 4.8**: Ο τελεστής ελέγχου ισότητας*

4.1.4.2 Ο τελεστής ανισότητας

Ο τελεστής ανισότητας (inequality operator) συμβολίζεται με “!=”. Ο τελεστής λειτουργεί με ανάλογο τρόπο και περιορισμούς με τον τελεστή ισότητας. Ισχύει !(a==b) == (a!=b), δηλαδή αν αντιστρέψουμε τον έλεγχο ισότητας μεταξύ δύο τελεστέων θα λάβουμε το ίδιο αποτέλεσμα με αυτό που θα λάβουμε εφαρμόζοντας τον τελεστή ανισότητας στους ίδιους τελεστέους. Επομένως, οι παρατηρήσεις σχετικά με τους πραγματικούς αριθμούς και τις μεταβλητές αναφοράς που επισημάνθηκαν στον τελεστή ισότητας ισχύουν κατ’ αναλογία και εδώ.

4.1.4.3 Οι υπόλοιποι σχεσιακοί τελεστές

Πρόκειται για τον τελεστή μεγαλύτερο από (greater than) που συμβολίζεται με τον χαρακτήρα ‘>’, τον τελεστή μεγαλύτερο ή ίσο από (greater than or equal to) που συμβολίζεται με τον χαρακτήρα ‘>=’, τον τελεστή μικρότερο από (less than) που συμβολίζεται με τον χαρακτήρα ‘<’ τον τελεστή μικρότερο ή ίσο από (less than or equal to) που συμβολίζεται με τον χαρακτήρα ‘<=’. Οι τελεστές αυτοί δέχονται ως τελεστέους όλους τους θεμελιώδεις τύπους πλην του τύπου boolean. Δεν συντάσσονται με μη θεμελιώδεις τύπους. Με πραγματικούς τελεστέους, θα πρέπει να χρησιμοποιούνται με την επιφύλαξη που προκύπτει από τον προσεγγιστικό χαρακτήρα των πραγματικών.

```
1     static void relationalOps() { //Κώδικας 4.9
2        int i = 5, j = 2;
3        char a = 'A', b = 'B';
4        System.out.println(i > j);
5        System.out.println(b > a);
6     }
```

***Κώδικας 4.9**: Ο τελεστής μεγαλύτερο από*

Η έξοδος της γραμμής 4 του κώδικα 4.9 εμφανίζει true. Στη γραμμή 5 συγκρίνονται δύο χαρακτήρες. Η έξοδος της γραμμής είναι επίσης true, καθώς εκείνο που συμβαίνει είναι πως συγκρίνονται οι κωδικοί των χαρακτήρων. Παρότι οι τελεστές αυτοί δεν συντάσσονται με String, αξίζει να σημειωθεί πως γενικότερα η σύγκριση μεταξύ αριθμών είναι αριθμητική, ενώ η σύγκριση μεταξύ σειρών χαρακτήρων είναι λεξικογραφική. Σύμφωνα με την αριθμητική σύγκριση, το 10 είναι μεγαλύτερο από το 2, ενώ σύμφωνα με τη λεξικογραφική το “10” είναι μικρότερο από το “2”.

4.1.5 Λογικοί τελεστές

Περιλαμβάνονται οι λογικοί τελεστές and και or, ο τριαδικός τελεστής και ο τελεστής instanceof. Ο τελεστής instanceof σχετίζεται με το αντικειμενοστρεφές μοντέλο και η παρουσίασή του γίνεται στην ενότητα 13.2.1.

4.1.5.1 Λογικό and

Το λογικό and συμβολίζεται ως "&&", είναι δυαδικός τελεστής και συντάσσεται αποκλειστικά με τελεστέους τύπου boolean. Η τιμή επιστροφής του υπολογίζεται όπως δείχνει ο πίνακας 4.2.

Τελεστέος 1	Τελεστέος 2	Αποτέλεσμα
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Πίνακας 4.2 *Πίνακας αληθείας λογικού and*

Η υλοποίηση του λογικού and ακολουθεί τη λογική βραχέως κυκλώματος (short circuit). Η αξιολόγηση γίνεται από αριστερά προς τα δεξιά. Αν ο πρώτος τελεστέος αρκεί για τον υπολογισμό του αποτελέσματος, ο δεύτερος τελεστέος δεν αξιολογείται. Σύμφωνα με τον πίνακα 2, όταν ο τελεστέος 1, έχει την τιμή false, τότε το αποτέλεσμα της έκφρασης είναι false ανεξάρτητα από την τιμή του δεύτερου τελεστέου. Αντίθετα, όταν ο πρώτος τελεστέος έχει την τιμή true, τότε το αποτέλεσμα της έκφρασης εξαρτάται από την τιμή του δεύτερου τελεστέου.

```
1    static void shortCircuitAnd() { //Κώδικας 4.10
2        int i = 1, j = 2;
3        System.out.println("i++>1 && j--<3 is " + (i++ > 1 && j-- < 3));
4        System.out.println("i= " + i + " j= " + j);
5        i = 1;
6        j = 2;
7        System.out.println("++i>1 && j--<3 is " + (++i > 1 && j-- < 3));
8        System.out.println("i= " + i + " j= " + j);
9    }
```

Κώδικας 4.10*: Short circuit and*

Η γραμμή 3 του κώδικα 4.10 τυπώνει i++>1 && j--<3 is false. Ο πρώτος τελεστέος είναι η λογική έκφραση i++>1, η τιμή του είναι false. Επομένως, ο δεύτερος, δηλαδή το j--<3 δεν θα αξιολογηθεί και άρα δεν θα μειωθεί η τιμή του j με αποτέλεσμα η έξοδος της γραμμής 4 είναι i= 2 j= 2. Αντίθετα, στη γραμμή 7, ο πρώτος τελεστέος του λογικού and είναι ++i>1 και η τιμή του είναι true. Επομένως θα αξιολογηθεί και ο δεύτερος τελεστέος με αποτέλεσμα να μειωθεί το j. Καθώς και ο δεύτερος τελεστέος είναι true, το αποτέλεσμα του λογικού and είναι επίσης true. Επομένως, η γραμμή 7 θα τυπώσει ++i>1 && j--<3 is true και η γραμμή 8 θα τυπώσει i= 2 j= 1.

4.1.5.2 Λογικό or

Το λογικό or συμβολίζεται ως "||", είναι δυαδικός τελεστής και συντάσσεται αποκλειστικά με τελεστέους τύπου boolean. Η τιμή επιστροφής του υπολογίζεται όπως δείχνει ο πίνακας 4.3.

Τελεστέος 1	Τελεστέος 2	Αποτέλεσμα
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Πίνακας 4.3 *Πίνακας αληθείας λογικού or*

Η υλοποίηση του λογικού or ακολουθεί επίσης τη λογική βραχέως κυκλώματος (short circuit). Η αξιολόγηση γίνεται από αριστερά προς τα δεξιά. Αν ο πρώτος τελεστέος αρκεί για τον υπολογισμό του αποτελέσματος, ο δεύτερος τελεστέος δεν αξιολογείται. Σύμφωνα με τον πίνακα 4.3, όταν ο τελεστέος 1, έχει την τιμή true, τότε το αποτέλεσμα της έκφρασης είναι true ανεξάρτητα από την τιμή του δεύτερου τελεστέου. Αντίθετα, όταν ο πρώτος τελεστέος έχει την τιμή false, τότε το αποτέλεσμα της έκφρασης εξαρτάται από την τιμή του δεύτερου τελεστέου.

```
1     static void shortCircuitOr() { //Κώδικας 4.11
2        int i = 1, j = 2;
3        System.out.println("i++>=1 || j--<3 is " + (i++ >= 1 || j-- < 3));
4        System.out.println("i= " + i + " j= " + j);
5        i = 1;
6        j = 2;
7        System.out.println("i++>=1 || j--<3 is " + (i++ > 1 || j-- < 3));
8       System.out.println("i= " + i + " j= " + j);
9     }
```

Κώδικας 4.11*: Short circuit or*

Η γραμμή 3 του κώδικα 4.11, τυπώνει i++>=1 || j--<3 is true. Ο πρώτος τελεστέος είναι η λογική έκφραση i++>=1, η τιμή του είναι true. Επομένως, ο δεύτερος, δηλαδή το j--<3 δεν θα αξιολογηθεί και άρα δεν θα μειωθεί η τιμή του j με αποτέλεσμα η έξοδος της γραμμής 4 να είναι i= 2 j= 2. Αντίθετα, στη γραμμή 7, ο πρώτος τελεστέος του λογικού or είναι ++i>1 και η τιμή του είναι false. Επομένως θα αξιολογηθεί και ο δεύτερος τελεστέος με αποτέλεσμα να μειωθεί το j. Καθώς ο δεύτερος τελεστέος είναι true, το αποτέλεσμα του λογικού or είναι επίσης true. Επομένως, η γραμμή 7 θα τυπώσει ++i>1 && j--<3 is true και η γραμμή 8 θα τυπώσει i= 2 j= 1.

4.1.5.3 Ο τριαδικός τελεστής

Ο τριαδικός τελεστής (ternary operator) συμβολίζεται ως "?:". Όπως φανερώνει το όνομά του, συντάσσεται με τρεις τελεστέους. Ο πρώτος είναι μια έκφραση τύπου boolean, π.χ. i<j, ο δεύτερος η τιμή που επιστρέφει ο τελεστής αν ο πρώτος τελεστέος έχει τιμή true και ο τρίτος τελεστέος δηλώνει την τιμή επιστροφής αν ο πρώτος τελεστέος έχει την τιμή false. Στον κώδικα 4.12 δίνουμε ένα παράδειγμα χρήσης του τριαδικού τελεστή. Στη γραμμή 3, ο πρώτος τελεστέος είναι η λογική έκφραση i<j. Στο συγκεκριμένο παράδειγμα, η έκφραση αυτή έχει τιμή true, επομένως επιστρέφεται η τιμή μετά το ?, δηλαδή η τιμή του i. Αν όμως η τιμή i<j ήταν false, τότε θα επιστρεφόταν ο τρίτος τελεστέος, δηλαδή το j, η τιμή μετά τον χαρακτήρα ':'.

```
1     static void ternaryOp() { //Κώδικας 4.12
2         int i = 10, j = 2;
3         int minIJ = i < j ? i : j;
4         System.out.println(minIJ);
5         double d = 5.2;
6         double max = i > d ? i : d;
7         System.out.println(max);
8     }
```

Κώδικας 4.12*: Ο τριαδικός τελεστής*

Ο τριαδικός τελεστής μπορεί να επιστρέψει δεδομένα οποιουδήποτε τύπου με τον περιορισμό πως οι τύποι του δεύτερου και τρίτου τελεστέου είναι συμβατοί μεταξύ τους. Αν οι τύποι αυτοί είναι ίδιοι μεταξύ τους, τότε ο τύπος επιστροφής του τελεστή είναι ίδιος με τον τύπο του δεύτερου και τρίτου τελεστέου. Αν δεν είναι ίδιοι, τότε ο τύπος επιστροφής είναι ο τύπος με τη μεγαλύτερη χωρητικότητα. Για παράδειγμα, στη γραμμή 6 του κώδικα 4.12, ο τελεστής επιστρέφει το i, τύπου int, εφόσον i>d και το d, τύπου double, εφόσον !(i>d). Ανεξάρτητα από ποια τιμή θα επιστραφεί, ο τύπος της θα είναι double.

4.1.6 Τελεστές ψηφίου

Υποστηρίζονται τέσσερις τελεστές ψηφίου, ο τελεστής ψηφίου AND, ο τελεστής ψηφίου OR, ο τελεστής ψηφίου XOR (Exclusive OR) και ο τελεστής συμπληρώματος. Οι τρεις πρώτοι είναι δυαδικοί, ενώ ο τελεστής συμπληρώματος είναι μοναδιαίος. Οι τελεστές ψηφίου δέχονται ως τελεστέους όλους τους ακέραιους τύπους και τον τύπο boolean. Οι ακέραιοι τύποι μπορούν να συνδυαστούν μεταξύ τους με τους τελεστές ψηφίου, όχι όμως και με τους boolean. Οι τελεστές ψηφίου όταν συντάσσονται με boolean λειτουργούν ως λογικοί τελεστές πλήρους αξιολόγησης. Οι τελεστές ψηφίου αντιμετωπίζουν τους τελεστέους τους σαν ψηφιοχάρτες (bitmaps) και κάνουν υπολογισμούς ψηφίο προς ψηφίο. Όταν συντάσσονται με ακέραιους τύπους, αν έστω και ένας από τους τελεστέους είναι long, τότε επιστρέφουν long αλλιώς int. Όταν συντάσσονται με boolean, επιστρέφουν boolean.

4.1.6.1 Τελεστής ψηφίου AND

Ο τελεστής ψηφίου AND (bitwise AND) συμβολίζεται με τον χαρακτήρα '&'. Ο τελεστής ακολουθεί τον πίνακα αληθείας του AND, δηλαδή αν και τα δύο ψηφία έχουν την τιμή 1, τότε το αντίστοιχο ψηφίο του αποτελέσματος έχει την τιμή 1. Αν έστω και ένα από τα ψηφία έχει την τιμή 0, τότε το αντίστοιχο ψηφίο του αποτελέσματος έχει την τιμή 0. Η γραμμή 4 του κώδικα 4.13 τυπώνει 0. Ο ψηφιοχάρτης του i αποτελείται μόνο από μηδενικά ψηφία. Τα μηδενικά αυτά ψηφία συνδυάζονται με τα αντίστοιχα ψηφία του j. Επομένως, όλα τα ψηφία του αποτελέσματος έχουν την τιμή 0.

```
1   static void bitwiseAnd() { //Κώδικας 4.13
2       int i = 0, j = 1232;
3       int k = i & j;
4       System.out.println("i&j=" + k);
5       int i1 = 0b00000000000000000000000000001010;
6       int j1 = 0b00000000000000011000011100001111;
7       System.out.println("i1=" + i1 + " " + "j1=" + j1);
8       System.out.println("i1&j1=" + (i1 & j1));
9       char d1 = 'A';
10      int m = i & d1;
11      System.out.println("i&d=" + m);
12      long b1 = 0;
13      long rslt = b1 & i;
14      System.out.println(rslt);
15  }
```

***Κώδικας 4.13**: Ο τελεστής ψηφίου AND*

Ένας άλλος τρόπος για να καταλάβουμε καλύτερα τους τελεστές ψηφίου είναι να ορίσουμε ακέραιες μεταβλητές σε δυαδική μορφή όπως έχουμε κάνει στις γραμμές 159 και 160. Στη γραμμή 7, τυπώνουμε τα i1 και j1 και βλέπουμε πως i1=10 και j1=100111; Το αποτέλεσμα του τελεστή μεταξύ των δύο αυτών μεταβλητών είναι ένας ψηφιοχάρτης που έχει 1 μόνο στις θέσεις που έχει 1 και η i1 και η j1. Όλες οι υπόλοιπες θέσεις καταλαμβάνονται από το 0. Οι θέσεις στις οποίες έχουν 1 και η i1 και η j1, ταυτίζονται με τις θέσεις που έχει 1 η i1. Επομένως, i1&j1==i1. Πράγματι, η γραμμή 8 τυπώνει i1&j1=10.

Όπως έχουμε ήδη αναφέρει, όλοι οι ακέραιοι τύποι είναι δεκτοί ως τελεστέοι. Στη γραμμή 10 εφαρμόζεται ο τελεστής μεταξύ μιας μεταβλητής τύπου int και μίας τύπου char. Το αποτέλεσμα είναι τύπου int και η τιμή του 0. Στη γραμμή 13 εφαρμόζεται ο τελεστής μεταξύ μιας μεταβλητής τύπου int και μιας τύπου long. Το αποτέλεσμα είναι τύπου long.

4.1.6.2 Τελεστής ψηφίου OR

Ο τελεστής ψηφίου OR (bitwise OR) συμβολίζεται με τον χαρακτήρα '|'. Ισχύουν και εδώ κατ' αναλογία, όσα παρατίθενται για τον τελεστή ψηφίου AND. Η διαφορά έγκειται στο ότι εδώ ακολουθείται ο πίνακας αληθείας του OR, δηλαδή αν ένα τουλάχιστον ψηφίο έχει την τιμή 1, τότε το αποτέλεσμα είναι 1, διαφορετικά είναι 0.

4.1.6.3 Τελεστής ψηφίου XOR:

Ο τελεστής ψηφίου XOR (bitwise XOR) συμβολίζεται με τον χαρακτήρα '^'. Ισχύουν και εδώ κατ' αναλογία όσα παρατίθενται για τον τελεστή ψηφίου AND. Η διαφορά έγκειται στο ότι εδώ ακολουθείται ο πίνακας αληθείας του XOR που παρατίθεται αμέσως από κάτω στον πίνακα 4.4, δηλαδή αν ένα μόνο ψηφίο έχει την τιμή 1, τότε το αποτέλεσμα είναι 1, διαφορετικά είναι 0.

bit 1	**bit 2**	**bit1^bit2**
1	1	0
1	0	1
0	1	1
0	0	0

Πίνακας 4.4 *Πίνακας XOR*

4.1.6.4 Τελεστής ψηφίου NOT

Ο τελεστής ψηφίου NOT (bitwise NOT) ή τελεστής συμπληρώματος (complement operator) είναι μοναδιαίος, συμβολίζεται με τον χαρακτήρα '~' και αντιστρέφει όλα τα bits του τελεστέου του. Ο τελεστής δεν συντάσσεται με μεταβλητές τύπου boolean.

4.1.7 Τελεστές διολίσθησης

Υποστηρίζονται οι τελεστές, διολίσθηση δεξιά, διολίσθηση αριστερά και μη προσημασμένη διολίσθηση δεξιά.

4.1.7.1 Διολίσθηση δεξιά

Ο τελεστής δεξιάς διολίσθησης (right shift) είναι δυαδικός, συμβολίζεται ως ">>"και συντάσσεται με ακέραιους τύπους. Εφόσον ο πρώτος τελεστέος του είναι long επιστρέφει long, διαφορετικά επιστρέφει int. Η εφαρμογή του τελεστή ενεργεί ως εξής: Καταρχάς δεν επηρεάζει τους τελεστέους αλλά δημιουργεί ένα αντίγραφο του ψηφιοχάρτη του πρώτου τελεστέου. Στη συνέχεια προκαλεί διολίσθηση προς τα δεξιά στο αντίγραφο του πρώτου τελεστέου κατά τον αριθμό των bits που καθορίζει η τιμή του δεύτερου τελεστέου. Έτσι, με τη διολίσθηση χάνεται ένας αριθμός από bits από τη δεξιά πλευρά του αντιγράφου. Τα bits που χάνονται αντικαθίστανται στην αριστερή πλευρά του αντιγράφου με τιμή την τιμή του πρώτου bit του, δηλαδή την τιμή του προσήμου. Ας σημειωθεί εδώ πως η Java χρησιμοποιεί για την κωδικοποίηση ακέραιων την παράσταση μέτρο και πρόσημο για τις θετικές τιμές και την παράσταση συμπληρώματος ως προς δύο (two's complement) για τις αρνητικές. Επομένως, η τιμή του προσήμου είναι 0 για τους θετικούς ακέραιους και 1 για τους αρνητικούς. Για λεπτομέρειες σχετικά με την κωδικοποίηση των ακέραιων σε μορφή συμπληρώματος ως προς δύο, ανατρέξτε στο παράρτημα 1.

```
  1   static void rightShift() { //Κώδικας 4.14
  2       int i = 3;
  3       int j = -3;
  4       System.out.println("Integer.toBinaryString(" + i + ")=" +
Integer.toBinaryString(i));
  5       System.out.println("Integer.toBinaryString(" + i + ">>1)=" +
Integer.toBinaryString(i >> 1));
  6       System.out.println("Integer.toBinaryString(" + j + ")=" +
Integer.toBinaryString(j));
  7       System.out.println("Integer.toBinaryString(" + j + ">>1)=" +
Integer.toBinaryString(j >> 1));
  8   }
```

Κώδικας 4.14*: Ο τελεστής δεξιάς διολίσθησης*

Στον κώδικα 4.14 έχουμε ορίσει τις ακέραιες μεταβλητές i και j. Στη γραμμή 4 εκτυπώνεται ο ψηφιοχάρτης της μεταβλητής i με τη βοήθεια της συνάρτησης toBinaryString. Η έξοδος αυτής της γραμμής είναι Integer.toBinaryString(3)=11. Πράγματι, η αναπαράσταση του 3 με τη μορφή συμπληρώματος του δύο είναι 11. Βέβαια, δεδομένου, ότι ο int στην Java έχει μέγεθος 32 bit, η πραγματική αναπαράσταση στη μνήμη είναι

00000000000000000000000000000011. Απλώς η Java δεν τυπώνει τυχόν μηδενικά στην αρχή ενός αριθμού (leading zeroes) δεδομένου ότι αυτά δεν επηρεάζουν την τιμή του. Η γραμμή 5 εκτυπώνει Integer.toBinaryString(3>>1)=1. Στη μεταβλητή i έγινε διολίσθηση δεξιά κατά 1 bit, επομένως από το αποτέλεσμα χάθηκε το τελευταίο 1, ενώ παράλληλα προστέθηκε στην αρχή ένα μηδέν. Ο πλήρης ψηφιοχάρτης του αποτελέσματος είναι 00000000000000000000000000000001.

Στη γραμμή 6 τυπώνεται ο ψηφιοχάρτης του -3. Πιο συγκεκριμένα, η έξοδος της γραμμής 6 είναι Integer.toBinaryString(-3)=11111111111111111111111111111101. Μετά τη διολίσθηση δεξιά κατά 1, το τελευταίο 1 χάνεται και προστίθεται στην αρχή του αποτελέσματος το ψηφίο 1. Έτσι, η έξοδος της γραμμής 7 είναι Integer.toBinaryString(-3>>1)=11111111111111111111111111111110. Ο ψηφιοχάρτης αυτός αντιστοιχεί στη δεκαδική τιμή -2.

4.1.7.2 Διολίσθηση αριστερά

Ο τελεστής αριστερής διολίσθησης (left shift) είναι δυαδικός, συμβολίζεται ως "<<"και συντάσσεται με ακέραιους τύπους. Η διαφορά του από τη διολίσθηση δεξιά είναι πως τα bits χάνονται από αριστερά και προστίθενται δεξιά. Επιπλέον, τα bits που προστίθενται έχουν την τιμή μηδέν ανεξάρτητα από το πρόσημο του πρώτου τελεστέου. Κατά τα λοιπά, ισχύουν κατ' αναλογία όσα ισχύουν και για τον τελεστή δεξιάς διολίσθησης.

```
    static void leftShift() { //Κώδικας 4.15
        int i = 3;
        int j = -3;
        System.out.println("Integer.toBinaryString(" + i + ")=       " +
Integer.toBinaryString(i));
        System.out.println("Integer.toBinaryString(" + i + "<<1)=" +
Integer.toBinaryString(i << 1));
        System.out.println("Integer.toBinaryString(" + j + ")=" +
Integer.toBinaryString(j));
        System.out.println("Integer.toBinaryString(" + j + "<<1)=" +
Integer.toBinaryString(j << 1));
        System.out.println("value of j<<1=" + (j << 1));
    }
```

***Κώδικας 4.15**: Τελεστής αριστερής διολίσθησης*

Η έξοδος του κώδικα 4.15 είναι η ακόλουθη:

```
Integer.toBinaryString(3)=11
Integer.toBinaryString(3<<1)= 110
Integer.toBinaryString(-3)=11111111111111111111111111111101
Integer.toBinaryString(-3<<1)= 11111111111111111111111111111010
value of j<<1=-6
```

Προσέξτε τον ψηφιοχάρτη του j, δηλαδή του -3. Μετά τη διολίσθηση έχει χαθεί ένα ψηφίο με τιμή 1 ενώ στο τέλος του ψηφιοχάρτη έχει προστεθεί ένα ψηφίο με τιμή 0. Η τιμή του j μετά τη διολίσθηση έχει γίνει -6.

4.1.7.3 Μη προσημασμένη διολίσθηση δεξιά

Ο τελεστής μη προσημασμένης δεξιάς διολίσθησης (unsigned right shift) συμβολίζεται με >>> και διαφέρει από τον τελεστή δεξιάς διολίσθησης κατά το ότι τα ψηφία που αντικαθιστούν όσα χάνονται δεν εξαρτώνται από το πρόσημο του πρώτου τελεστέου αλλά έχουν πάντα την τιμή 0.

Ας σημειωθεί πως στην Java δεν υπάρχει τελεστής μη προσημασμένης αριστερής διολίσθησης, καθώς αυτός θα ήταν πανομοιότυπος με την αριστερή διολίσθηση.

4.1.8 Σύνθετη εκχώρηση

Υποστηρίζεται μια σειρά δυαδικών τελεστών σύνθετης εκχώρησης (compound assignment), δηλαδή τελεστών που συνδυάζουν την εκχώρηση με μία άλλη πράξη. Στον πίνακα 4.5 παρουσιάζονται οι τελεστές σύνθετης εκχώρησης μαζί με την ισοδύναμη έκφραση που αντιστοιχεί στον κάθε ένα.

Τελεστής	**Παράδειγμα Χρήσης**	**Ισοδύναμη έκφραση**
+=	i+=5	i=i+5
-=	i-=k	i=i-k
*=	i*=k	i=i*k
/=	i/=k	i=i/k
%=	i%=k	i=i%k
&=	i&=k	i=i&k
|=	i|=k	i=i|k
<<=	i<<=k	i=i<<k
>>=	i>>=k	i=i>>k
>>>=	i>>>=k	i=i>>>k

Πίνακας 4.5 *Τελεστές Σύνθετης Εκχώρησης*

4.2. Βασική Διαμόρφωση εξόδου

Συχνά χρειάζεται να διαμορφώσουμε τα δεδομένα πριν τα εμφανίσουμε. Για παράδειγμα, σε μια εφαρμογή μπορεί να επιθυμούμε να εμφανίσουμε τους πραγματικούς αριθμούς με δύο δεκαδικά ψηφία. Σε αυτήν την ενότητα παρουσιάζονται δύο εναλλακτικοί τρόποι για τη διαμόρφωση της εξόδου.

4.2.1 Η printf

Η printf αποτελεί τον παραδοσιακό τρόπο διαμόρφωσης της εξόδου με την έννοια ότι υποστηρίζεται και στην C [2]. Ας ξεκινήσουμε την παρουσίαση με ένα παράδειγμα:

```
1   static void stringF() { ////Κώδικας 4.16
2       System.out.printf("%s%n", "Hello");
3       System.out.printf("%-10s%10S%n", "Hello", "Hello");
4       System.out.printf("%10.3S%n", "Hello");
5   }
```

Κώδικας 4.16*: Διαμόρφωση εξόδου του τύπου String*

Η πρώτη παράμετρος σε όλες τις κλήσεις της printf στον κώδικα 4.16 ονομάζεται προσδιοριστής διαμόρφωσης (format specifier). Ο προσδιοριστής διαμόρφωσης περιλαμβάνει μια σειρά από κανόνες διαμόρφωσης (formatting rules). Κάθε κανόνας σηματοδοτεί την αρχή του με τον χαρακτήρα %. Στη γραμμή 2, ο προσδιοριστής διαμόρφωσης έχει την τιμή %s%n. Επομένως περιλαμβάνει δύο κανόνες: Ο πρώτος κανόνας εκφράζεται με τον χαρακτήρα s και μας λέει πως η διαμόρφωση αφορά αλφαριθμητική σειρά. Ο χαρακτήρας s σε αυτό το πλαίσιο ονομάζεται χαρακτήρας μετατροπής (conversion character). Άλλοι χαρακτήρες μετατροπής είναι ο d για διαμόρφωση ακέραιων, ο f για διαμόρφωση πραγματικών αριθμών, ο c για διαμόρφωση χαρακτήρων και ο b για διαμόρφωση τιμών boolean. Ωστόσο, οι χαρακτήρες μετατροπής c και b χρησιμοποιούνται σπάνια. Ο δεύτερος κανόνας περιλαμβάνει τον χαρακτήρα n που προκαλεί αλλαγή γραμμής.

Η δεύτερη παράμετρος στη γραμμή 2 είναι το String που επιθυμούμε να τυπώσουμε. Καθότι ο κανόνας διαμόρφωσης στην πρώτη γραμμή δεν καθορίζει τίποτα περισσότερο παρά μόνο ότι θα εμφανίσουμε μια αλφαριθμητική σειρά και θα προβούμε σε αλλαγή γραμμής, εκείνο που θα δούμε ως έξοδο είναι απλώς η σειρά Hello.

Στη γραμμή 3, μετά τον προσδιοριστή διαμόρφωσης έχουμε μια σειρά από String. Είναι γενικό χαρακτηριστικό της printf ότι μετά τον προσδιοριστή διαμόρφωσης μπορεί να δέχεται πολλές παραμέτρους. Για περισσότερες πληροφορίες επάνω σε αυτό, δείτε την ενότητα 7.3.3. Κάθε μία από αυτές τις παραμέτρους

προορίζεται για να εμφανιστεί. Τη διαμόρφωσή τους ρυθμίζει ο προσδιοριστής διαμόρφωσης, δηλαδή η πρώτη παράμετρος. Στη γραμμή 3 λοιπόν έχουμε αυτόν τον προσδιοριστή προσπέλασης %-10s%10S%n. Περιλαμβάνει τρεις κανόνες. Ο τρίτος, όπως εξηγήθηκε παραπάνω, προκαλεί αλλαγή γραμμής της εξόδου. Ο πρώτος αφορά τη δεύτερη παράμετρο της printf και ο δεύτερος την τρίτη. Οι δύο πρώτοι κανόνες περιλαμβάνουν το 10. Αυτό προσδιορίζει το πλάτος στο οποίο θα εμφανιστεί η αντίστοιχη παράμετρος. Τα δεδομένα στοιχίζονται σε αυτό το πλάτος στη δεξιά πλευρά, δηλαδή το Hello σε πλάτος 10 θα εμφανιστεί ως ␣␣␣␣␣Hello, που ␣ συμβολίζει ένα διάστημα (space). Αν ωστόσο επιθυμούμε τα δεδομένα μας να στοιχηθούν στα αριστερά του πλάτους, τότε τοποθετούμε τον χαρακτήρα – (μείον) όπως έχουμε κάνει στον πρώτο κανόνα. Σημειώστε πως σε αυτό το πλαίσιο ο χαρακτήρας μείον ονομάζεται σηματοφόρος (flag). Έτσι το πρώτο Hello θα εμφανισθεί ως Hello␣␣␣␣␣. Αν επιθυμούμε η σειρά να εμφανιστεί με κεφαλαία, τότε ως χαρακτήρα διαμόρφωσης δίνουμε το S αντί του s, όπως κάνουμε στον δεύτερο κανόνα. Στη γραμμή 4 το πλάτος ακολουθείται από την ακρίβεια (precision). Η ακρίβεια δηλώνεται με μια τελεία ακολουθούμενη από έναν ακέραιο.
Η έξοδος του κώδικα 4.16 έχει ως εξής:

```
Hello
Hello          HELLO
       HEL
```

Το γενικό συντακτικό ενός κανόνα διαμόρφωσης έχει ως εξής:

```
%[σηματοφορείς][πλάτος][.ακρίβεια]χαρακτήρας μετατροπής
```

Από αυτά τα στοιχεία, μόνο ο χαρακτήρας % που δηλώνει την αρχή του κανόνα και ο χαρακτήρας μετατροπής που δηλώνει τον τύπο των δεδομένων ή αλλαγή γραμμής είναι υποχρεωτικά. Οι σηματοφορείς, το πλάτος και η ακρίβεια είναι προαιρετικά στοιχεία. Ωστόσο, η σειρά με την οποία τα στοιχεία τοποθετούνται μέσα σε έναν κανόνα είναι υποχρεωτικά και αυτή δείχνει το γενικό συντακτικό του κανόνα.
Ας δούμε και μερικά άλλα χρήσιμα παραδείγματα:

```
1   static void intF() { //Κώδικας 4.17
2       System.out.printf("ακέραιος: %d%n", 1000000L);
3       System.out.printf("ακέραιος: %,d%n", 1000000L);
4       System.out.printf(Locale.ENGLISH,"ακέραιος: %,d%n", 1000000L);
5   }
```

Κώδικας 4.17*: Διαμόρφωση εξόδου ακέραιων τύπων*

Όπως φαίνεται στον κώδικα 4.17, ένας προσδιοριστής διαμόρφωσης μπορεί να περιλαμβάνει και απλές αλφαριθμητικές σειρές οι οποίες απλώς αποτελούν μέρος της εξόδου. Στη γραμμή 2 δεν δίνουμε κάποια ιδιαίτερη διαμόρφωση, οπότε η έξοδος αυτής της γραμμής είναι απλά 1000000. Στη γραμμή 3 όμως, έχουμε τοποθετήσει τον σηματοφόρο (κόμμα). Με αυτόν τον σηματοφόρο ζητάμε ο αριθμός να εμφανιστεί με διαχωριστικά χιλιάδων, οπότε η έξοδος εδώ θα είναι 1.000.000. Επειδή όμως τα διαχωριστικά χιλιάδων δεν είναι ίδια για όλες τις χώρες, μας δίνεται η δυνατότητα να χρησιμοποιήσουμε μια άλλη έκδοση της printf που μας επιτρέπει να καθορίσουμε ποιο ακριβώς διαχωριστικό θα χρησιμοποιήσουμε. Έτσι, η γραμμή 4 εμφανίζει 1,000,000.

```
1     static void realF() { //Κώδικας 4.18
2       System.out.printf("%f%n", 5.1473);
3       System.out.printf("%(010.2f%n", -5.1473);
4       System.out.printf("%-10.2f%n", 5.1473);
5   }
```

Κώδικας 4.18*: Διαμόρφωση εξόδου πραγματικών τύπων*

Η έξοδος του κώδικα 4.18 έχει ως εξής:

```
5,147300
```

```
(00005,15)
5,15
```

Στη γραμμή 3, ο σηματοφόρος '(' (αριστερή παρένθεση) έχει ως αποτέλεσμα να εμφανίζονται οι αρνητικοί αριθμοί εντός παρενθέσεων. Το δε 0 που ακολουθεί καλύπτει το πλάτος με μηδενικά στα αριστερά. Ο σηματοφόρος 0 δεν συντάσσεται με στοίχιση στα αριστερά. Κατά τα λοιπά, με όσα έχουμε συζητήσει μέχρι εδώ, είναι εύκολο να ερμηνεύσετε την έξοδο του κώδικα 4.18.

4.2.2 Η DecimalFormat

Ένας εναλλακτικός τρόπος για διαμόρφωση δεκαδικών αριθμών για έξοδο βασίζεται στην κλάση DecimalFormat. Η DecimalFormat παρέχει ευκολίες στη διαμόρφωση των αριθμών για έξοδο, εισάγει ωστόσο κάποιον βαθμό πολυπλοκότητας στον κώδικά μας.

```
1    static void outputFormat() { //Κώδικας 4.19
2        DecimalFormat f = new DecimalFormat("###,###.##");
3        double d1 = 9.566, d2 = 9.273;
4        System.out.println(f.format(d1) + "+" + f.format(d2) + " = " +
f.format(d1 + d2));
5
6        d1 = 9.565;
7        d2 = 9.275;
8        System.out.println(f.format(d1) + "+" + f.format(d2) + " = " +
f.format(d1 + d2));
9
10        d1 = 0.275;
11        d2 = 0.385;
12        System.out.println(f.format(d1) + "+" + f.format(d2) + " = " +
f.format(d1 + d2));
13
14        d1 = 9_856_327.563;
15        d2 = 5627869.289;
16        System.out.println(f.format(d1) + "+" + f.format(d2) + " = " +
f.format(d1 + d2));
17
18        f.applyPattern("###,###.###");
19        System.out.println(f.format(d1) + "+" + f.format(d2) + " = " +
f.format(d1 + d2));
20
21        f.applyPattern("€###,###.###");
22        System.out.println(f.format(d1));

23        f.applyPattern("(€###,###.###)");
24        System.out.println(f.format(d1));
25        int k = 128;
26        System.out.println(f.format(k));
27
28        char c = 'A';
29        System.out.println(f.format(c));
30
31        f.applyPattern("000,000,000.0000");
32        System.out.println(f.format(d1));
33    }
```

***Κώδικας 4.19**: Διαμόρφωση εξόδου*

Η έξοδος της outputFormat είναι η εξής:

```
9,57+9,27 = 18,84
9,56+9,28 = 18,84
```

```
0,28+0,39 = 0,66
9.856.327,56+5.627.869,29 = 15.484.196,85
9.856.327,563+5.627.869,289 = 15.484.196,852
€9.856.327,563
(€9.856.327,563)
(€128)
(€65)
009.856.327,5630
```

Η διαμόρφωση βασίζεται στην προκαθορισμένη κλάση DecimalFormat [2]. Για να τη χρησιμοποιήσετε προσθέστε στην κλάση σας την import Java.text.DecimalFormat.

Στη γραμμή 2 του κώδικα 4.19 δημιουργούμε ένα αντικείμενο της κλάσης DecimalFormat. Η μεταβλητή f είναι μεταβλητή αναφοράς. Κατά τη δημιουργία της συνδέεται με το μοτίβο (pattern), "###,###.##". Στη γραμμή 4 στέλνουμε προς εμφάνιση στην οθόνη τρεις πραγματικές τιμές, χρησιμοποιώντας τη συνάρτηση format της DecimalFormat. Η συνάρτηση format επιστρέφει μια αλφαριθμητική σειρά. Η σειρά που επιστρέφει η format υπολογίζεται με βάση την τιμή που της δίνουμε κατά την κλήση της και το μοτίβο που ορίσαμε κατά τη δημιουργία της f. Στο μοτίβο αυτό ο χαρακτήρας ',' συμβολίζει τον διαχωριστή χιλιάδων, ο χαρακτήρας '.' συμβολίζει τον διαχωριστή δεκαδικών και ο χαρακτήρας '#' συμβολίζει το ψηφίο. Επομένως, σύμφωνα με το μοτίβο, ζητάμε μια διαμόρφωση στην οποία να εμφανίζονται τα ψηφία του αριθμού κατά τρόπο που οι χιλιάδες να διαχωρίζονται μεταξύ τους, όπως επίσης και να διαχωρίζεται το δεκαδικό από το ακέραιο μέρος και το δεκαδικό μέρος να εμφανίζεται με δύο ψηφία. Η έξοδος της γραμμής 4 είναι 9.57+9.27 = 18.84. Προσέξτε καταρχάς πως έχουμε στρογγυλοποίηση του δεκαδικού μέρους. Η μεταβλητή d1 εμφανίστηκε ως 9.57 αντί του 9.566. Αντίστοιχα, η d2 εμφανίζεται ως 9.27 αντί του 9.273. Η format στρογγυλοποιεί εξ ορισμού τα δεκαδικά προς την πλησιέστερη τιμή. Το 0.57 είναι πλησιέστερο στο 0.566 σε σχέση με το 0.56. Παρόμοια, το 0.27 είναι πλησιέστερο στο 0.273 από ότι το 0.28.

Όταν όμως δεν υπάρχει πλησιέστερη τιμή, τότε η στρογγυλοποίηση γίνεται προς τον άρτιο γείτονα. Η έξοδος της γραμμής 8 είναι 9.56+9.28 = 18.84. Προσέξτε πως η τιμή του d1 από 9.565 μετατράπηκε σε 9.56. Πράγματι, η τιμή 9.565 απέχει εξίσου από το 9.56 και το 9.57. Σε αυτήν την περίπτωση η format στρογγυλοποίησε προς τον άρτιο γείτονα, δηλαδή το 6 και όχι τον περιττό, δηλαδή το 7. Αντίστοιχα, η d2 με τιμή 9.275, εμφανίζεται ως 9.28. Επομένως, η στρογγυλοποίηση έγινε προς τον άρτιο γείτονα, το 8, και όχι τον περιττό, δηλαδή το 7.

Η έξοδος της γραμμής 12, μετά τη μεταβολή των τιμών των d1 και d2 που έγινε στη γραμμή 10 και 11, είναι 0.28+0.39 = 0.66. Επομένως, το d1 με τιμή 0.275 εμφανίζεται ως 0.28 και το d2 με τιμή 0.385 εμφανίζεται ως 0.39. Εκείνο όμως που χρήζει ιδιαίτερης προσοχής είναι πως το άθροισμα 0.28+0.39 ισούται με 0.67 και όχι με 0.66. Η ασυμφωνία αυτή προκύπτει καθώς υπολογίζουμε πρώτα το άθροισμα d1+d2 και στη συνέχεια το στρογγυλοποιούμε. Επομένως, 0.275+0.385=0.610. Η στρογγυλοποίηση στα δύο δεκαδικά του 0.610 είναι προφανώς το 0.61. Είναι κατανοητό από αυτό το παράδειγμα πως η στρογγυλοποίηση κατά την εμφάνιση πραγματικών αριθμών πρέπει να γίνεται με ιδιαίτερη προσοχή.

Στη γραμμή 16 το μοτίβο "###,###.##" εφαρμόζεται σε αριθμούς με περισσότερα ψηφία από τους χαρακτήρες του χωρίς να δημιουργείται κάποιο πρόβλημα.

Στη γραμμή 18 αλλάζουμε το μοτίβο της f. Το νέο μοτίβο υποστηρίζει τρία δεκαδικά ψηφία. Έτσι η έξοδος της γραμμής 19 είναι 9,856,327.563+5,627,869.289 = 15,484,196.852.

Είναι εφικτό κατά τη διαμόρφωση των αριθμών να προσθέτουμε σταθερούς χαρακτήρες. Έτσι στη γραμμή 21 έχουμε προσθέσει το σύμβολο του Ευρώ ώστε να προηγείται της αριθμητικής τιμής. Πράγματι η έξοδος της γραμμής 22 είναι €9,856,327.563.

Στη γραμμή 23 τροποποιούμε πάλι το μοτίβο της f ώστε να περικλείεται ο αριθμός και το σύμβολο του Ευρώ από παρενθέσεις. Έτσι, η έξοδος της γραμμής 24 είναι (€9,856,327.563).

Στη γραμμή 26 επιδεικνύεται πώς το μοτίβο μπορεί να εφαρμοστεί και σε ακέραιους. Η έξοδος της γραμμής είναι (€128). Παρότι το μοτίβο προβλέπει δεκαδικά ψηφία, η format τυπώνει σωστά τους ακέραιους τύπους χωρίς δεκαδικά.

Στη γραμμή 29 εκτυπώνεται ο χαρακτήρας 'A' μέσω της format. Η έξοδος της γραμμής είναι (€65), που σημαίνει πως στη θέση του χαρακτήρα εμφανίζεται ο κωδικός του.

Σε κάποιες εφαρμογές θέλουμε να εμφανίζουμε αριθμούς σε σταθερό πλάτος συμπληρώνοντας τα στοιχεία που λείπουν με μηδενικά. Σε αυτήν την περίπτωση, προκειμένου να μην αλλοιωθεί η τιμή του

αριθμού, μηδενικά μπορούν να προστεθούν μόνο στην αρχή του ακέραιου μέρους ή στο τέλος του δεκαδικού μέρους ενός αριθμού. Στη γραμμή 31 μεταβάλλουμε το μοτίβο, έτσι ώστε ο αριθμός να τοποθετείται σε πλάτος 13 ψηφίων από τα οποία τα 9 ανήκουν στο ακέραιο μέρος και τα 4 στο δεκαδικό. Η έξοδος της γραμμής 32 είναι 009,856,327.5630.

Ας σημειωθεί πως η κλάση DecimalFormat παρέχει πολλές επιπλέον δυνατότητες. Για παράδειγμα, τη δυνατότητα να καθορίσουμε ως διαχωριστικό χιλιάδων τον χαρακτήρα '.' και ως διαχωριστικό δεκαδικών τον χαρακτήρα ',' ή να αλλάξουμε τον τρόπο στρογγυλοποίησης. Περισσότερες πληροφορίες για την DecimalFormat μπορείτε να βρείτε στην τεκμηρίωση της Oracle [1].

Επιπλέον, υπάρχουν και άλλοι τρόποι για τη διαμόρφωση της εξόδου όπως η συνάρτηση format της κλάσης String [3].

4.3 Είσοδος από το πληκτρολόγιο

Για είσοδο από το πληκτρολόγιο θα χρησιμοποιούμε την προκαθορισμένη κλάση Scanner. Για να χρησιμοποιηθεί πρέπει να την εισάγετε στον κώδικά σας προσθέτοντας την import Java.util.Scanner. Με την Scanner μπορούμε να διαβάσουμε δεδομένα και από αρχεία. Ωστόσο σε αυτήν την ενότητα, περιοριζόμαστε στη χρήση της Scanner για απλή είσοδο από το πληκτρολόγιο. Η ανάγνωση αρχείων παρουσιάζεται αναλυτικά στην ενότητα 15.

Η Scanner για κάθε θεμελιώδη τύπο διαθέτει μια συνάρτηση με την οποία μπορούμε να διαβάζουμε αντίστοιχα δεδομένα. Ο πίνακας 4.6 παρουσιάζει τις συναρτήσεις που μας ενδιαφέρουν ώστε να μπορούμε να υποστηρίζουμε στα προγράμματά μας είσοδο από το πληκτρολόγιο.

Τύπος επιστροφής	**Συνάρτηση**	**Χρήση**
String	next()	Είσοδος δεδομένων τύπου String
boolean	nextBoolean()	Είσοδος δεδομένων τύπου boolean
byte	nextByte()	Είσοδος δεδομένων τύπου byte
double	nextDouble()	Είσοδος δεδομένων τύπου double
float	nextFloat()	Είσοδος δεδομένων τύπου float
int	nextInt()	Είσοδος δεδομένων τύπου int
String	nextLine()	Είσοδος δεδομένων τύπου String
long	nextLong()	Είσοδος δεδομένων τύπου long
short	nextShort()	Είσοδος δεδομένων τύπου short

Πίνακας 4.6 *Συναρτήσεις της Scanner για είσοδο δεδομένων*

Σε κάθε κλήση τους, κάθε μια από τις συναρτήσεις του πίνακα 4.6 διαβάζει μια ολοκληρωμένη σειρά δεδομένων (token). Εξ ορισμού, μια σειρά δεδομένων θεωρείται ότι τερματίζεται από την παρουσία ενός διαστήματος (whitespace). Εξαίρεση αποτελεί η συνάρτηση nextLine που διαβάζει τα δεδομένα μιας γραμμής, θεωρώντας το διάστημα ως μέρος των δεδομένων.

Στον κώδικα 4.20 επιδεικνύεται ενδεικτική χρήση των συναρτήσεων εισόδου. Στη γραμμή 2, ορίζουμε το αντικείμενο in της κλάσης Scanner μέσα από το οποίο έχουμε τη δυνατότητα να καλούμε τις συναρτήσεις εισόδου. Κατά τον ορισμό του in, το συνδέουμε με την System.in. Με αυτόν τον τρόπο καθορίζεται πως η είσοδος θα γίνεται από το πληκτρολόγιο.

```
1     static void keyboardInput() { //Κώδικας 4.20
2        Scanner in = new Scanner(System.in);
3        System.out.print("Δώστε όνομα : ");
4        String name = in.nextLine();
5
6        System.out.print("Δώστε αριθμό Μητρώου: ");
7        int aem = in.nextInt();
8
9        System.out.print("Εισόδημα έτους 2021 : ");
10       double eisodima = in.nextDouble();
11       System.out.println();
```

```
12
13          System.out.println("Ονομα: " + name);
14          System.out.println("AM : " + aem);
15          System.out.println("Εισόδημα 2021: " + eisodima);
16         // in.close();
17      }
```

Κώδικας 4.20*: Είσοδος από το πληκτρολόγιο*

Στη γραμμή 4 διαβάζουμε μια γραμμή δεδομένων. Θα μπορούσαμε να χρησιμοποιήσουμε τη συνάρτηση next(), μιας και το όνομα είναι τύπου String. Ωστόσο, αν ο χρήστης καταχωρήσει όνομα και επώνυμο διαχωρισμένα με διάστημα, η next() θα διαβάσει μόνο μέχρι το διάστημα, δηλαδή μόνο το όνομα. Ακόμη χειρότερα, η επόμενη είσοδος που επιχειρείται με την nextInt στη γραμμή 7 θα βρει ως δεδομένα εισόδου το επώνυμο του χρήστη. Θα βρει δηλαδή ένα String, ενώ περιμένει int, οπότε θα παραχθεί λάθος χρόνου εκτέλεσης.

Εφόσον διαβάσουμε σωστά το ονοματεπώνυμο του χρήστη, προχωράμε στην είσοδο του αριθμού μητρώου. Αν εδώ ο χρήστης δεν δώσει ακέραια τιμή, επίσης θα παραχθεί λάθος χρόνου εκτέλεσης. Στη συνέχεια, στη γραμμή 10, διαβάζουμε έναν αριθμό τύπου double.

Το παράδειγμα κλείνει με την in.close() σχολιασμένη. Προσοχή, μην κλείσετε ποτέ ένα αντικείμενο Scanner που είναι σχετισμένο με την System.in. Αν το κάνετε, η εφαρμογή σας δεν θα είναι σε θέση να ανοίξει ροή επικοινωνίας με το πληκτρολόγιο παρά μόνο αν επανεκκινηθεί [3].

Βέβαια, η σωστή χρήση της Scanner απαιτεί τη διαχείριση των λαθών που ενδέχεται να παραχθούν, ζήτημα με το οποίο ασχολούμαστε στην ενότητα 16.

4.4 Ασκήσεις

1. Ποια είναι η έξοδος του κώδικα που ακολουθεί;

```
int i = 0, j = 0;
System.out.print(i++ + " " + (++j) + " ");
System.out.println(i == j);
```

2. Ποια είναι η έξοδος του κώδικα που ακολουθεί; Εξηγήστε την απάντησή σας.

```
int k = 2;
System.out.println((k += 2) == (k *= 2));
```

3. Ποια είναι η έξοδος του κώδικα;

```
int i,j,k;
i = j = k = 0;
System.out.println((i++ == 0 || j++ == 0) && k++ == 0);
System.out.println(i + " " + j + " " + k);
```

4. Ποια είναι η έξοδος του κώδικα;

```
int i,j,k;
i = j = k = 0;
System.out.println(i++ == 0 || j++ == 0 && k++ == 0);
System.out.println("i++ == 0 || j++ == 0 && k++ == 0");
System.out.println(i + " " + j + " " + k);
```

5. Ποια είναι η έξοδος του κώδικα;

```
int i,j,k;
i = j = k = 0;
```

```
System.out.println(i++ == 0 && j++ == 0 || k++ == 0);
System.out.println("i++ == 0 && j++ == 0 || k++ == 0");
System.out.println(i + " " + j + " " + k);
```

6. Λάβετε από το πληκτρολόγιο 2 πραγματικούς αριθμούς και εμφανίστε στην οθόνη το γινόμενό τους. Εμφανίστε τα αποτελέσματα κατά τρόπο ώστε να προηγείται η περιγραφή τους και με 2 δεκαδικά ψηφία κατά μέγιστο.

```
Παράδειγμα
Δώσε αριθμό: 3.58
Δώσε αριθμό: 2.99
Το γινόμενο 3.58 x 2.99 = 10.7
```

7. Χρησιμοποιήστε μια μόνο println για να τυπώστε τη διεύθυνσή σας με την ακόλουθη μορφή:

```
Ονοματεπώνυμο
Οδός Αριθμός
Πόλη, ΤΚ
Χώρα
```

Υπόδειξη: Θυμηθείτε τους χαρακτήρες διαφυγής στην ενότητα 3.3.

8. Ο Γιώργος λαμβάνει μηνιαίο μισθό 1000 Ευρώ. Επίσης, λαμβάνει ενοίκιο από το διαμέρισμα στην Καλλιθέα 450 Ευρώ τον μήνα και το διπλάσιο ενοίκιο από το διαμέρισμα στου Παπάγου. Ο Γιώργος έχει ένα επιπλέον μηνιαίο εισόδημα. Τα έξοδα του μηνός Απριλίου έχουν ως εξής: Για τρόφιμα ξόδεψε τα 12/100 του συνολικού του εισοδήματος, τα καύσιμα κοστίζουν 0.2 Ευρώ το χιλιόμετρο, το Ηλεκτρικό ρεύμα 200 Ευρώ και το κόστος νερού 120 Ευρώ. Να αναπτυχθεί αλγόριθμος που ζητά από τον Γιώργο το επιπλέον μηνιαίο εισόδημα και τα χιλιόμετρα του Απριλίου και εμφανίζει κατάσταση εσόδων-εξόδων του Απριλίου σύμφωνα με το παρακάτω υπόδειγμα:

```
--------Κατάσταση Εσόδων-Εξόδων Απριλίου-----------------
Μισθός 1000
Ενοίκιο Καλλιθέας 450
Ενοίκιο Παπάγου 900
Επιπλέον μηνιαίο εισόδημα 100
-----------------------------------------------------------
Συνολικό Εισόδημα 2450

Κόστος Τροφίμων 294.00
Κόστος Καυσίμων 160.00
Κόστος Ηλεκτρικού Ρεύματος 160.00
Κόστος Νερού 120
-----------------------------------------------------------
Συνολικό Κόστος 774.00
***********************************************************
Διαφορά 1676.00
```

9. Ο Γιάννης είναι καπνιστής. Κάθε μέρα καταναλώνει ένα πακέτο τσιγάρα που κοστίζει 4 Ευρώ +20% ΦΠΑ. Επίσης, κάθε δύο μήνες χρειάζεται έναν αναπτήρα. Ρωτήστε τον χρήστη πόσο κοστίζει ένας αναπτήρας και υπολογίστε πόσο κοστίζει στον Γιάννη το κάπνισμα κάθε ημέρα και πόσο κάθε μήνα. Ποιο είναι το ετήσιο κόστος; Θεωρήστε πως όλοι οι μήνες αποτελούνται από 30 ημέρες. Για να υπολογίσετε το ημερήσιο κόστος επιμερίστε το κόστος του αναπτήρα στις ημέρες χρήσης του. Αν το ετήσιο εισόδημα του Γιάννη είναι 8.000 Ευρώ, πόσο τοις εκατό του εισοδήματός του ξοδεύει για κάπνισμα; Αν τα υπόλοιπα μηνιαία έξοδα του Γιάννη είναι 500 Ευρώ, τι ποσό από το εισόδημά του αποταμιεύει κάθε χρόνο;

Βιβλιογραφία

[1] “Operators (The Java™ Tutorials > Learning the Java Language > Language Basics).” https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html (accessed Sep. 14, 2021).

[2] “Formatting Numeric Print Output (The Java™ Tutorials > Learning the Java Language > Numbers and Strings).” https://docs.oracle.com/javase/tutorial/java/data/numberformat.html (accessed Sep. 14, 2021).

[3] K. Fogel, “If You’re Using Java’s Scanner Class for Keyboard Input, You’re Doing it Wrong - DZone Java,” DZone. https://dzone.com/articles/if-youre-using-javas-scanner-class-for-keyboard-in (accessed Sep. 14, 2021).

Κεφάλαιο 5

Σύνοψη
Στην ενότητα αυτή παρουσιάζονται οι δομές επιλογής και οι επαναληπτικές δομές της Java. Πιο συγκεκριμένα, παρουσιάζονται οι δομές επιλογής, if, if…else, switch και ο τριαδικός τελεστής. Επίσης, παρουσιάζονται οι επαναληπτικές δομές for, while, do-while και οι λέξεις-κλειδιά break και continue. Περιλαμβάνονται λυμένες ασκήσεις και ασκήσεις προς λύση.

Προαπαιτούμενη γνώση
Οι θεμελιώδεις τύποι και βασική γνώση του τύπου String. Βασική είσοδος και έξοδος. Η λειτουργία των τελεστών.

Λέξεις κλειδιά
Δομή επιλογής, επαναληπτική διαδικασία, βρόχος, τριαδικός τελεστής

5 Έλεγχος Ροής

Ο έλεγχος της ροής του προγράμματος πραγματοποιείται με τις δομές επιλογής και τις επαναληπτικές δομές. Οι δομές επιλογής μας δίνουν τη δυνατότητα να δημιουργούμε πολλαπλές διαδρομές μέσα στον κώδικά μας και να επιλέγουμε κάθε φορά την καταλληλότερη ανάλογα με τις συνθήκες που επικρατούν κατά την εκτέλεση του προγράμματος. Με τις επαναληπτικές δομές ρυθμίζουμε την επανάληψη τμημάτων του κώδικά μας, επίσης, ανάλογα με συγκεκριμένες συνθήκες που επικρατούν κατά την εκτέλεση του προγράμματος. Τόσο οι δομές επιλογής όσο και οι επαναληπτικές δομές αποτελούν βασικές δομές προγραμματισμού, όχι μόνο στην Java αλλά σε κάθε προγραμματιστικό περιβάλλον.
Επιπλέον, τόσο οι δομές επιλογής όσο και οι επαναληπτικές δομές βασίζονται στον έλεγχο μιας συνθήκης (condition). Η συνθήκη είναι μία λογική έκφραση, δηλαδή μία έκφραση τύπου boolean. Για παράδειγμα, η έκφραση i>5 && j<0 είναι λογική έκφραση, καθώς λαμβάνει την τιμή true ή false ανάλογα με την τιμή των i και j.

5.1 Δομές επιλογής

Οι δομές επιλογής μας δίνουν τη δυνατότητα να καθορίζουμε διαφορετικές διαδρομές που ακολουθεί ο κώδικάς μας ανάλογα με τις συνθήκες που επικρατούν στο πρόγραμμά μας κατά τον χρόνο εκτέλεσης. Πρόκειται για θεμελιώδεις δομές που συναντώνται με τη μία ή την άλλη μορφή σε κάθε γλώσσα προγραμματισμού. Για να αντιληφθεί κανείς τη σπουδαιότητά τους αρκεί να αναλογιστεί πως χωρίς αυτές τα προγράμματα θα ήταν μια συλλογή εντολών που θα εκτελούνταν αναγκαστικά σειριακά. Με άλλα λόγια, η λήψη αποφάσεων κατά την εκτέλεση του προγράμματος θα ήταν αδύνατη. Για παράδειγμα, αν ένα πρόγραμμα διέθετε ως δεδομένα μια σειρά από ονόματα και κάθε όνομα συνδεόταν με χαρακτηρισμό φύλου, δεν θα ήταν σε θέση να προσφωνήσει τους άντρες ως κυρίους και τις γυναίκες ως κυρίες.

Η Java υποστηρίζει τις δομές επιλογής if, if…else, switch και τον τριαδικό τελεστή. Είναι όμως εφικτό να συνδυαστούν οι δομές if και if…else διαμορφώνοντας έτσι μια ιδιαίτερη δομή γνωστή ως εμφωλευμένη (nested) if. Στη συνέχεια παρουσιάζονται μια προς μια οι δομές επιλογής της Java.

5.1.1 Η επιλογή if

Η γενική μορφή της if δίνεται ακολούθως:

```
if (condition) {
     statement(s);
}
```

Η δεσμευμένη λέξη if ακολουθείται από τη συνθήκη ελέγχου μέσα σε παρενθέσεις. Αν η συνθήκη είναι αληθής, η ροή του προγράμματος μπαίνει στο μπλοκ που ακολουθεί και εκτελεί τα statements. Αν η συνθήκη είναι ψευδής, η ροή του προγράμματος μεταπηδά στην έξοδο από το μπλοκ της if.

Μέσα στο μπλοκ της if μπορούμε να έχουμε ένα ή περισσότερα statements. Αν έχουμε μόνο ένα, οι αγκύλες είναι προαιρετικές, ωστόσο, συνηθίζεται να τοποθετούνται.

```
static void simpleIf() { //Κώδικας 5.1
    int i = 1, j = -1;
    if (i > 0 && j < 1) {
        System.out.println("i μεγαλύτερο από το 0");
        System.out.println("j μικρότερο από το 1");
    }
    System.out.println("Μετά τον έλεγχο της έκφρασης \"i>0 && j<1\"");
}
```

Κώδικας 5.1 *Παράδειγμα απλής if*

Στον κώδικα 5.1 η τιμή της συνθήκης ελέγχου εξαρτάται από τις τιμές των i και j. Αν για παράδειγμα, i==1 και j==-1, η συνθήκη είναι αληθής, οπότε οι προτάσεις "i μεγαλύτερο από το 0" και "j μικρότερο από το 1" θα εμφανιστούν. Αν όμως το i είναι μικρότερο ή ίσο με το 0 ή το j είναι μεγαλύτερο ή ίσο με το 1, η συνθήκη είναι ψευδής, οπότε αυτές οι προτάσεις δεν θα εμφανιστούν. Σε κάθε περίπτωση, θα τυπωθεί η πρόταση "Μετά τον έλεγχο της έκφρασης "i>0 && j<1"".

```
static void IfWithSimpleStatement() { //Κώδικας 5.2
    int i = 0;
    if (i > 0) {
        System.out.println("Το i είναι μεγαλύτερο από το 0");
    }
    System.out.println("Μετά τον έλεγχο της έκφρασης \"i>0\"");
}
```

Κώδικας 5.2 *Παράδειγμα απλής if με ένα statement*

Ο κώδικας 5.2 παρουσιάζει μια if που εφόσον η συνθήκη της αληθεύει εκτελεί μια μόνο πρόταση. Σε αυτήν την περίπτωση δεν είναι υποχρεωτικό να χρησιμοποιηθούν αγκύλες. Ωστόσο, στην πράξη, συχνά τοποθετούνται αγκύλες από πολλούς επαγγελματίες αλλά και από πολλά IDE, ακόμη και στην περίπτωση μίας πρότασης.
Η πρόταση που ακολουθεί μια if, μπορεί να είναι και η ίδια μια if. Έτσι προκύπτει μια δομή γνωστή ως εμφωλευμένη if (nested if).

```
static void nestedIf() { //Κώδικας 5.3
    int i = 1, j = -1;
    if (i > 0) {
        if (j < 1) {
            System.out.println("i>0 και j<1");
        }
    }
}
```

Κώδικας 5.3 *Παράδειγμα εμφωλευμένης if*

Ο κώδικας 5.3 ελέγχει καταρχάς αν το i είναι μεγαλύτερο του μηδενός. Αν η συνθήκη αληθεύει, προχωρά στην εκτέλεση της επόμενης πρότασης που είναι επίσης μια if. Η δεύτερη if ελέγχει αν το j είναι μικρότερο από το 1. Εφόσον η συνθήκη αυτή αληθεύει, τότε τυπώνει τη σειρά χαρακτήρων "i>0 και j<1".

5.1.2 Η επιλογή if…else

Η γενική μορφή της if…else έχει ως εξής:

```
if (condition) {
```

```
        statement(s);
    }
    else {
        statement(s);
    }
```

Στην περίπτωση που η συνθήκη αληθεύει εκτελούνται οι προτάσεις μέσα στο μπλοκ της if. Αν η συνθήκη είναι ψευδής, εκτελούνται οι προτάσεις μέσα στο μπλοκ της else. Τόσο οι προτάσεις μέσα στο μπλοκ της if όσο και οι προτάσεις στο μπλοκ της else μπορεί να αρχίζουν επίσης με μια if.

```
static void ifElseDemo() { //Κώδικας 5.4
    int i = 1, j = 2;
    if (i > 0) {
        if (j < 2 && j > 0) {
            System.out.println("j ίσο με 1");
        } else if (j < 2) {
            System.out.println("j<=0");
        }
        System.out.println("i>0");
    } else {
        System.out.println("i<=0");
    }
}
```

Κώδικας 5.4 *Παράδειγμα if…else*

Στον κώδικα 5.4 ελέγχεται καταρχάς η τιμή του i. Αν αυτή είναι θετική, η ροή συνεχίζει στο μπλοκ της εξωτερικής if, διαφορετικά η ροή διακλαδώνεται στον όρο else στο τέλος του κώδικα όπου τυπώνεται η σειρά "i<=0". Στην περίπτωση που η ροή εισέλθει στο μπλοκ της εξωτερικής if, ελέγχεται το j. Εφόσον η μεταβλητή j είναι ακέραιου τύπου και το j είναι μικρότερο του 2 και μεγαλύτερο του 0, δεν μπορεί παρά να είναι 1, οπότε τυπώνεται η σειρά "j ίσο με 1". Αν όμως η συνθήκη j < 2 && j>0 είναι ψευδής, δηλαδή το j δεν είναι 1, τότε ελέγχεται αν το j είναι μικρότερο του 2. Εφόσον το j είναι μικρότερο του 2 και δεν είναι 1, σημαίνει πως είναι μικρότερο ή ίσο του 0, οπότε τυπώνεται η σειρά "j<=0". Σε κάθε περίπτωση, μέσα στο μπλοκ της εξωτερικής if εκτυπώνεται η σειρά "i>0", ενώ στον όρο else της εξωτερικής if τυπώνεται η σειρά "i<=0".

5.1.3 Η switch

Σε μερικές περιπτώσεις ο έλεγχος της ροής του προγράμματος καθορίζεται από πολλαπλές συνθήκες. Όλες αυτές οι περιπτώσεις μπορούν να κωδικοποιηθούν με χρήση εμφωλευμένων if. Ωστόσο, ανάλογα με το πλήθος των συνθηκών, η χρήση εμφωλευμένων if αυξάνει την πολυπλοκότητα και καθιστά τον κώδικα δυσανάγνωστο. Σε κάποιες από αυτές τις περιπτώσεις ο κώδικας μπορεί να απλοποιηθεί με χρήση της switch. Η γενική μορφή της switch έχει ως εξής:

```
switch(expression) {
    case constant1: statement(s); break;
    case constant2: statement(s); break;
    case constant3: statement(s); break;
    …
    …
    default: statement(s);
}
```

Η expression της switch πρέπει να είναι τύπου char, byte, short, int ή String. Οι τιμές ελέγχου στους όρους case πρέπει να είναι σταθερές, δηλαδή literals ή final μεταβλητές. Τα statement(s) όπως και στην if αντιπροσωπεύουν ένα ή περισσότερα statements. Ωστόσο εδώ στην περίπτωση που έχουμε περισσότερα από ένα statements δεν είναι υποχρεωτικό να τα εσωκλείσουμε σε αγκύλες. Η break ακολουθεί τα statements μίας

case. Ωστόσο, η παρουσία της είναι προαιρετική ανάλογα με το αποτέλεσμα που θέλουμε να πετύχουμε. Πάντως, η break έχει ως αποτέλεσμα την έξοδο από την switch. Η default είναι επίσης προαιρετική.

Πώς δουλεύει όμως η switch; Καταρχάς αξιολογείται η expression. Ανάλογα με την τιμή της, ο έλεγχος της ροής κατευθύνεται στην κατάλληλη case. Αν σε καμία case δεν προβλέπεται η τιμή της expression, ο έλεγχος της ροής κατευθύνεται στην default εφόσον υπάρχει αλλιώς έξω από την switch. Στην περίπτωση που ταιριάζει η τιμή της expression με την τιμή ελέγχου μιας case, εκτελούνται τόσο τα statements της αντιστοιχιζόμενης case όσο και τα statements των επόμενων case ή και της default έως ότου εκτελεστεί ένα break.

Η default δεν τοποθετείται υποχρεωτικά ως τελευταίος όρος της switch. Αντίθετα, μπορεί να τοποθετηθεί ως πρώτος όρος ή οπουδήποτε ανάμεσα στις case. Χρειάζεται όμως προσοχή ως προς το πού θα τοποθετήσουμε μια default πρόταση, γιατί σε συνδυασμό με τα break μπορεί να έχουμε εντελώς διαφορετικά αποτελέσματα.

Αποσαφηνίζουμε παρακάτω με παραδείγματα χρήσης της switch:

```
static void switchDemo() { //Κώδικας 5.5
    int cV = 2;
    final int con = 3;
    switch (cV) {
        case 1:
            System.out.print("1");
            System.out.println("-case 1");
            break;
        case 2:
            System.out.println("2");
            break;
        case con:
            System.out.println("three");
            break;
        default:
            System.out.println("default");
    }
}
```

Κώδικας 5.5 *Τυπικό παράδειγμα χρήσης της switch*

Έστω στον κώδικα 5.5, η μεταβλητή cV είναι ακέραιου τύπου. Αν η τιμή της είναι 1, η έξοδος του κώδικα θα είναι 1-case 1, δηλαδή η τιμή της μεταβλητής αντιστοιχίζεται στην case 1 και επομένως εκτελούνται οι τρεις statements της case 1. Το πρώτο statement τυπώνει 1, το δεύτερο -case 1, ενώ το τρίτο statement είναι ένα break που τερματίζει την switch. Αν η cV έχει την τιμή 2, τότε τυπώνεται το 2. Αν η τιμή της cV είναι 3, τότε τυπώνεται η σειρά "three". Προσέξτε πως αν η μεταβλητή con δεν είχε δηλωθεί ως final, ο μεταγλωττιστής θα παρήγαγε κατάλληλο λάθος.

Τέλος, αν η τιμή της cV είναι διαφορετική και από το 1 και από το 2 και από το 3, τότε εκτελείται η default και εκτυπώνεται η σειρά “default”. Σημειώστε πως η default δεν ακολουθείται από break. Πράγματι, η τοποθέτηση ενός break στον τελευταίο όρο μιας switch δεν μεταβάλλει κατά κανένα τρόπο το αποτέλεσμα της switch, καθώς μετά τον τελευταίο όρο ακολουθεί έτσι κι αλλιώς έξοδος από την switch.

```
static void switchNoBreak() { //Κώδικας 5.6
    int cV = 1;
    final int con = 3;
    switch (cV) {
        case 1:
            System.out.print("1");
        case 2:
            System.out.print("2");
        case con:
            System.out.print("3");
        default:
            System.out.println("default");
    }
```

```
    }
```

Κώδικας 5.6 *Παράδειγμα switch χωρίς break*

Αν η τιμή της cV είναι 1, η έξοδος του κώδικα 5.6 είναι 123default. Αν η τιμή της είναι 2, η έξοδος του κώδικα είναι 23default. Αν η τιμή της είναι 3, η έξοδος είναι 3default. Τέλος, αν η τιμή της cV είναι διάφορη και από το 1 και από το 2 και από το 3, η έξοδος είναι default. Επομένως, αν η break απουσιάζει, τότε η switch κατευθύνεται στην κατάλληλη case και από εκτελεί όλα τα statements έως ότου βρει break ή έως ότου τερματίσει η switch.
Όπως αναφέραμε προηγουμένως, η default δεν τοποθετείται υποχρεωτικά ως τελευταίος όρος μιας switch. Ωστόσο, η συμπεριφορά της switch δεν είναι ανεξάρτητη από τη θέση της default.

```
    static void switchDefault() { //Κώδικας 5.7
        int cV = 1;
        final int con = 3;
        switch (cV) {
            default:
                System.out.print("default");
            case 1:
                System.out.print("1");
            case 2:
                System.out.print("2");
            case con:
                System.out.println("3");
        }
    }
```

Κώδικας 5.7 *Παράδειγμα switch με την default ως πρώτο όρο*

Αν η τιμή της cV είναι 1, η έξοδος του κώδικα 5.7 είναι 123. Αν η τιμή της είναι 2, η έξοδος του κώδικα είναι 23. Αν η τιμή της είναι 3, η έξοδος είναι 3. Τέλος, αν η τιμή της cV είναι διάφορη και από το 1 και από το 2 και από το 3, η έξοδος είναι default123.
Ας δούμε και ένα παράδειγμα όπου η expression είναι τύπου String.

```
    static void switchString() { //Κώδικας 5.8
        String monthLabel = "Jan";
        String monthAA = null;
        switch (monthLabel) {
            case "Jan":
                monthAA = "first";
                break;
            case "Feb":
                monthAA = "second";
                break;
            case "Mar":
                monthAA = "third";
                break;
            case "Apr":
                monthAA = "fourth";
                break;
            case "May":
                monthAA = "fifth";
                break;
            case "Jun":
                monthAA = "sixth";
                break;
            case "Jul":
                monthAA = "seventh";
                break;
            case "Aug":
```

```
                monthAA = "eighth";
                break;
            case "Sep":
                monthAA = "ninth";
                break;
            case "Oct":
                monthAA = "tenth";
                break;
            case "Nov":
                monthAA = "eleventh";
                break;
            case "Dec":
                monthAA = "twelfth";
        }
        if (monthAA == null) {
            System.out.println("Unkown Month Name: " + monthLabel);
        } else {
            System.out.println(monthLabel + " is the " + monthAA + " month of the
year");
        }
    }
```

Κώδικας 5.8 *Παράδειγμα switch με έκφραση ελέγχου τύπου String*

5.1.4 Ο τριαδικός τελεστής

Ο τριαδικός τελεστής που είδαμε ήδη στην ενότητα 4.1.5.3, αποτελεί μια συνοπτική δομή επιλογής. Παρουσιάζει μεγάλη ομοιότητα με τη δομή επιλογής if…else. Για παράδειγμα, ο κώδικας 5.9, επιτυγχάνει δύο φορές το ίδιο αποτέλεσμα, την πρώτη χρησιμοποιεί τον τριαδικό τελεστή και τη δεύτερη την δομή if…else.

```
    static void ternaryOp() { //Κώδικας 5.9
        int i = 1, j = 2, max;

        max = i > j ? i : j;

        if (i > j) {
            max = i;
        } else {
            max = j;
        }
    }
```

Κώδικας 5.9 *Παράδειγμα αναλογίας τριαδικού τελεστή και if…else*

5.2 Επαναληπτικές δομές

Οι επαναληπτικές δομές αποτελούν παρόμοιες με τις δομές επιλογής θεμελιώδεις δομές κάθε γλώσσας προγραμματισμού. Αναλογιστείτε πως χωρίς αυτές αν επιθυμούσαμε να τυπώσουμε τους ακέραιους από το 1 έως το 100.000, θα έπρεπε να πληκτρολογήσουμε 100.000 εντολές. Στην ουσία, ο αποτελεσματικός προγραμματισμός των Ηλεκτρονικών Υπολογιστών θα ήταν αδύνατος. Στα αγγλικά, η επαναληπτική δομή αναφέρεται με τον όρο loop που αποδίδεται στα ελληνικά ως βρόχος. Οι επαναληπτικές δομές που υποστηρίζει η Java είναι η while, η do-while και η for.

5.2.1 Η επαναληπτική δομή while

Η γενική μορφή της επαναληπτικής δομής while έχει ως εξής:

```
while(condition) {
     statement(s);
}
```

Η while εξετάζει τη συνθήκη condition και όσο αυτή είναι αληθής εκτελεί μια ή περισσότερες statements. Παρόμοια με την if, αν έχουμε μια μόνο statement, οι αγκύλες είναι προαιρετικές.

```
static void whileDemo() { //Κώδικας 5.10
    int i = 10;
    while (i > 0) {
        System.out.print("i>0 ");
        i--;
    }
    System.out.println("i=" + i);
}
```

Κώδικας 5.10 *Παράδειγμα απλής while*

Στον κώδικα 5.10 η ακέραιη μεταβλητή i αρχικοποιείται στην τιμή 10. Στη συνέχεια η while εξετάζει αν το i είναι μεγαλύτερο από το 0. Μέσα στο μπλοκ της while τυπώνεται το μήνυμα "i>0" και ακολούθως μειώνεται η τιμή του i κατά 1. Η διαδικασία επαναλαμβάνεται έως ότου το i πάψει να είναι μεγαλύτερο από το 0. Δεδομένου ότι το i αρχικοποιήθηκε στο 10 και η τιμή του μειώνεται κατά 1 σε κάθε επαναληπτικό βήμα, το i θα γίνει 0 ακριβώς μετά από 10 επαναλήψεις. Επομένως, ο κώδικας 5.10, τυπώνει 10 φορές τη σειρά "i>0", στη συνέχεια το i γίνεται 0, η συνθήκη i>0 καθίσταται ψευδής και ο κώδικας εξέρχεται από τον βρόχο while, οπότε εκτελεί την επόμενη πρόταση και τυπώνει i=0.

Προσέξτε πως αν στον κώδικα 5.10 η μεταβλητή i είχε αρχικοποιηθεί σε τιμή μικρότερη ή ίση με το 0, η συνθήκη ελέγχου θα ήταν εξαρχής ψευδής και ο κώδικας δεν θα εισερχόταν ποτέ στο μπλοκ της while. Αντίθετα, αν η μεταβλητή i δεν μειωνόταν μέσα στην while, τότε η ροή του κώδικα δεν θα εξερχόταν από τον βρόχο. Σε αυτήν την περίπτωση λέμε ότι έχουμε έναν ατέρμονο βρόχο (endless loop). Προφανώς, η παρουσία ατέρμονου βρόχου μέσα σε ένα πρόγραμμα αποτελεί σοβαρό προγραμματιστικό λάθος.

Τόσο η while όσο και οι υπόλοιπες επαναληπτικές δομές μπορούν να αξιοποιήσουν τις λέξεις-κλειδιά break και continue. Η μεν break μεταφέρει τη ροή του προγράμματος στην πρώτη πρόταση έξω από το μπλοκ της while, η δε continue μεταφέρει τη ροή στη συνθήκη ελέγχου.

Ας υποθέσουμε πως θέλουμε να τυπώσουμε 10 το πολύ τυχαίες ακέραιες τιμές στην κλίμακα 1 έως 10. Αν όμως κάποια από τις τιμές αυτές είναι μικρότερη από το 2, τότε να την τυπώσουμε και να σταματήσουμε ακόμη και αν δεν έχει συμπληρωθεί ο αριθμός των 10 εκτυπωμένων τιμών.
Ένας τρόπος για να πετύχουμε αυτήν την απαίτηση δίνεται στον κώδικα 5.11.

```
static void prtRandoms() {
    int i = 0, rI;
    Random generator = new Random();
    while (i < 10) {
        i++;
        rI = generator.nextInt(10) + 1;
        if (rI >= 2) {
            System.out.println(rI);
        } else {
            System.out.println(rI);
            break;
        }
    }
}
```

Κώδικας 5.11 *Παράδειγμα while με break*

Καταρχάς στον κώδικα 5.11 δηλώνουμε και αρχικοποιούμε τη μεταβλητή i στο 0. Παράλληλα δηλώνουμε τη μεταβλητή rI. Στη συνέχεια δημιουργούμε το αντικείμενο generator. Πρόκειται για μια γεννήτρια τυχαίων αριθμών. Σημειώστε πως για να χρησιμοποιήσετε ένα τέτοιο αντικείμενο θα πρέπει να εισάγετε στο πρόγραμμά σας τη βιβλιοθήκη Java.util.Random. Αυτό επιτυγχάνεται αν εισάγετε αμέσως μετά τη δήλωση

του πακέτου τη σειρά import Java.util.Random; Στη συνέχεια αναπτύσσουμε μια επαναληπτική διαδικασία while. Η συνθήκη ελέγχου είναι i<10, δηλαδή η επαναληπτική διαδικασία θα επαναλαμβάνεται όσο το i θα είναι μικρότερο από το 10. Δεδομένου ότι το i αρχικοποιήθηκε στο 0 και σε κάθε επανάληψη αυξάνεται κατά 1, θα εκτελεστούν 10 το πολύ επαναλήψεις. Αυτό είναι συμβατό με την απαίτηση να τυπώσουμε 10 το πολύ τυχαίους ακέραιους. Στη συνέχεια χρησιμοποιούμε τον generator για να παράξουμε έναν τυχαίο αριθμό. Η συνάρτηση nextInt παράγει έναν τυχαίο ακέραιο από το 0 έως την τιμή του ορίσματός της μείον 1, δηλαδή από το 0 έως το 9. Σε αυτόν τον αριθμό προσθέτουμε το 1, οπότε ο ακέραιος που εκχωρείται στη μεταβλητή rI είναι ένας τυχαίος από το 1 έως το 10. Μετά ελέγχουμε την τιμή της rI και αν είναι μεγαλύτερη ή ίση με το 2, απλώς την τυπώνουμε διαφορετικά, ισχύει η απαίτηση σύμφωνα με την οποία αν κάποια από τις τυχαίες τιμές είναι μικρότερη από το 2, τότε θα πρέπει να την τυπώσουμε και να σταματήσουμε περαιτέρω εκτύπωση τυχαίων αριθμών. Αυτό ακριβώς κάνει ο κώδικας 5.11. Στη συνέχεια τυπώνουμε την τιμή που είναι μικρότερη από το 2 και αμέσως μετά εκτελείται η break που διακόπτει την επαναληπτική διαδικασία και μεταφέρει τη ροή έξω από την επαναληπτική διαδικασία όπου ακολουθεί και η έξοδος της συνάρτησης prtRandoms.

Δύο σχόλια σχετικά με τον κώδικα 5.11. Γενικά είναι καλό στα προγράμματά μας να αποφεύγουμε τη φλυαρία, δηλαδή όπου είναι εφικτό να προτιμούμε τον βραχύτερο έναντι του μακρύτερου κώδικα. Προσέξτε πως ο κώδικας System.out.println(rI) εκτελείται και στις δύο διακλαδώσεις της if, δηλαδή είτε ισχύει η συνθήκη rI>=2 είτε όχι. Ένας τέτοιος κώδικας μπορεί να βγει πάντα έξω από την if έτσι ώστε μια παρουσία του να είναι αρκετή.

Ένα άλλο θέμα που θα μπορούσαμε να βελτιώσουμε σχετίζεται με την αύξηση του i. Αφού ελέγξουμε το i στην επόμενη γραμμή, το αυξάνουμε κατά 1. Το ίδιο ακριβώς αποτέλεσμα μπορούμε να πετύχουμε αν αυξήσουμε το i στη συνθήκη ελέγχου με χρήση του μεταθεματικού τελεστή προσαύξησης. Προσοχή όμως! Αν η αύξηση γίνει όπως γίνεται στον κώδικα 5.11, δεν έχει ιδιαίτερη σημασία αν χρησιμοποιηθεί ο προθεματικός ή ο μεταθεματικός τελεστής προσαύξησης. Αν όμως η αύξηση γίνει μέσα στη συνθήκη, τότε η χρήση του μεταθεματικού τελεστή διαφοροποιεί ουσιαστικά τη συμπεριφορά του κώδικα. Μελετήστε πώς την αλλάζει και γιατί. Ο κώδικας 5.12 παράγει το ίδιο ακριβώς αποτέλεσμα με τον κώδικα 5.11 με το πλεονέκτημα ότι είναι βραχύτερος.

```
static void prtRandoms() { //Κώδικας 5.12
    int i = 0, rI;
    Random generator = new Random();
    while (i++ < 10) {
        rI = generator.nextInt(10) + 1;
        System.out.println(rI);
        if (rI < 2) {
            break;
        }
    }
}
```

Κώδικας 5.12 *Βραχύτερη έκδοση της συνάρτησης prtRandoms*

Ας υποθέσουμε τώρα πως θέλουμε να τυπώσουμε 10 τυχαίες ακέραιες τιμές στην κλίμακα 1 έως 10 εξαιρώντας τις τιμές 3 και 4, δηλαδή 10 τιμές που ανήκουν στο σύνολο {1..10}-{3,4}. Η λέξη-κλειδί continue θα μας βοηθήσει στην επίλυση αυτού του προβλήματος.

```
static void prt10Randoms() { //Κώδικας 5.13
    int i = 0, rI;
    Random generator = new Random();
    while (i < 10) {
        rI = generator.nextInt(10) + 1;
        if (rI == 3 || rI == 4) {
            continue;
        }
        System.out.println(rI);
        i++;
    }
}
```

Κώδικας 5.13 *Παράδειγμα while με χρήση της λέξης-κλειδί continue*

Μέσα στον βρόχο while του κώδικα 5.13 παράγουμε καταρχάς έναν τυχαίο αριθμό από 1 έως 10. Στη συνέχεια ελέγχουμε αν πρόκειται για αριθμό που εξαιρείται. Αν όντως πρόκειται για αριθμό που εξαιρείται, καλούμε την continue ώστε να μεταφέρουμε τη ροή στην αρχή του βρόχου όπου ελέγχεται η συνθήκη i<10. Αν δεν πρόκειται για αριθμό που εξαιρείται, προχωρούμε κανονικά στην εκτύπωσή του και στη συνέχεια αυξάνουμε τη μεταβλητή ελέγχου i κατά 1. Προσοχή, εδώ η i θα ήταν λάθος να αυξηθεί μέσα στη συνθήκη ελέγχου. Κάτι τέτοιο θα είχε ως αποτέλεσμα την παραγωγή συνολικά 10 ακέραιων και την εκτύπωση μόνο όσον είναι διάφοροι από το 3 και το 4, δηλαδή να εκτυπωθούν πιθανότατα λιγότεροι από 10 ακέραιους που προσδιορίζουν οι απαιτήσεις του προβλήματός μας.

5.2.2 Η επαναληπτική δομή do-while

Η γενική μορφή της do-while έχει ως εξής:

```
do {
     statement(s);
} while (condition);
```

Η διαφορά της από την while είναι πως η συνθήκη ελέγχεται στο τέλος. Επομένως, η do-while εγγυάται πως ο κώδικας μέσα στο σώμα της θα εκτελεστεί τουλάχιστον μία φορά.

Ας υποθέσουμε πως θέλουμε να λάβουμε μια σειρά από ακέραιους από τον χρήστη και να υπολογίσουμε το άθροισμά τους και επιπλέον, να σταματήσουμε τη διαδικασία εισόδου από τον χρήστη όταν αυτός δώσει την τιμή 0.

Σε αυτό το πρόβλημα, ο βρόχος do-while θα μας φανεί πιο χρήσιμος από τον βρόχο while, καθώς είναι αναγκαίο να λάβουμε μια τουλάχιστον είσοδο από τον χρήστη.

```
static void calcSum() { //Κώδικας 5.14
    int i;
    int sum = 0;
    Scanner in = new Scanner(System.in);
    do {
        System.out.print("Enter int (0 terminates input):");
        i = in.nextInt();
        sum += i;
    } while (i != 0);
    System.out.println("Sum = " + sum);
}
```

Κώδικας 5.14 *Παράδειγμα χρήσης του βρόχου do-while*

Στον κώδικα 5.14 ορίζουμε τη μεταβλητή i που θα χρησιμοποιήσουμε για να καταχωρούμε την είσοδο του χρήστη. Τη μεταβλητή i δεν είναι αναγκαίο να την αρχικοποιήσουμε, καθώς είναι βέβαιο πως πριν χρησιμοποιηθεί θα εκχωρηθεί τιμή σε αυτήν. Στη συνέχεια ορίζουμε τη μεταβλητή sum. Είναι αναγκαίο να αρχικοποιήσουμε την sum στο 0, καθώς αυτή προσαυξάνεται κατά την είσοδο του χρήστη σε κάθε βήμα της επαναληπτικής διαδικασίας do-while. Ακολούθως ορίζουμε ένα αντικείμενο της κλάσης Scanner για να το χρησιμοποιήσουμε για είσοδο από τον χρήστη. Στο επόμενο βήμα μπαίνουμε στην επαναληπτική διαδικασία do-while όπου ζητάμε και παίρνουμε έναν ακέραιο από τον χρήστη με τον οποίο ενημερώνουμε τον αθροιστή sum. Στο τέλος της do-while γίνεται ο έλεγχος της συνθήκης. Αν ο χρήστης έδωσε ακέραιο διάφορο του 0, η διαδικασία επαναλαμβάνεται αλλιώς τερματίζεται.

Όλες οι επαναληπτικές διαδικασίες μπορούν να υλοποιηθούν με χρήση οποιασδήποτε επαναληπτικής δομής. Ωστόσο, κάποιες δομές παρέχουν διευκολύνσεις ανάλογα με τις απαιτήσεις που πρέπει να υλοποιηθούν. Στον κώδικα 5.15 επιλύουμε το ίδιο πρόβλημα με αυτό του κώδικα 5.14 αλλά χρησιμοποιώντας τη δομή while αντί της δομής do-while.

```
static void calcSum() { //Κώδικας 5.15
    int i;
```

```
        int sum = 0;
        Scanner in = new Scanner(System.in);
        System.out.print("Enter int (0 terminates input):");
        i = in.nextInt();
        while (i != 0) {
            sum += i;
            System.out.print("Enter int (0 terminates input):");
            i = in.nextInt();
        }
        System.out.println("Sum = " + sum);
    }
```

Κώδικας 5.15 *Υλοποίηση της calcSum με while αντί do-while*

Προσέξτε πως στον κώδικα 5.15 οι εντολές εισόδου από τον χρήστη εμφανίζονται δύο φορές, μία έξω από τον βρόχο και μια μέσα στον βρόχο μετά την επεξεργασία που στην περίπτωσή μας είναι η πρόσθεση του ακέραιου i στον αθροιστή sum. Πρόκειται για μια τυπική μεταφορά ενός do-while σε while. Ωστόσο, η διπλή παρουσία των εντολών εισόδου μπορεί να αποφευχθεί αν οργανώσουμε τον κώδικά μας όπως φαίνεται στον κώδικα 5.16.

```
    static void calcSum() { //Κώδικας 5.16
        int i;
        int sum = 0;
        Scanner in = new Scanner(System.in);
        while (true) {
            System.out.print("Enter int (0 terminates input):");
            i = in.nextInt();
            if (i == 0) {
                break;
            }
            sum += i;
        }
        System.out.println("Sum = " + sum);
    }
```

Κώδικας 5.16 *Προσομοίωση της do-while με while*

Στον κώδικα 5.16 προσομοιώνουμε τη λειτουργία της do-while με χρήση της while. Η ροή του κώδικα θα εισέλθει οπωσδήποτε μια τουλάχιστον φορά στον βρόχο while, καθώς η συνθήκη ελέγχου είναι πάντα true. Στην ουσία ο έλεγχος του βρόχου γίνεται από την if, η οποία αν εντοπίσει τιμή ίση με το 0, καλεί την break που προκαλεί έξοδο από τον βρόχο.
Συχνά δημιουργείται η αντίληψη ότι η continue κατευθύνει τη ροή του προγράμματος στην αρχή της επαναληπτικής διαδικασίας. Η αντίληψη αυτή όμως είναι λανθασμένη. Η continue κατευθύνει τη ροή στη συνθήκη ελέγχου του βρόχου. Αν ο βρόχος είναι τύπου while, τότε η αρχή της διαδικασίας συμπίπτει με τη συνθήκη ελέγχου. Αν όμως ο βρόχος είναι τύπου do-while, τότε η συνθήκη βρίσκεται στο τέλος. Ο κώδικας 5.17 επιδεικνύει τη λειτουργία της continue σε έναν do-while βρόχο.

```
    static void checkContinue() {
        int i = 0;
        do {
            i++;
            System.out.println(i);
            if (i == 1) {
                continue;
            }
            System.out.println(i);
        } while (i < 1);
    }
```

Κώδικας 5.17 *Η λειτουργία της continue*

Ο κώδικας 5.17 όταν εκτελείται τυπώνει 1. Το αποτέλεσμα του κώδικα είναι απόδειξη ότι η continue μεταφέρει τη ροή στη συνθήκη ελέγχου και όχι απλώς στην αρχή του βρόχου. Με την είσοδο μέσα στον βρόχο, η μεταβλητή i αυξάνεται κατά 1 και γίνεται 1 από 0 που ήταν η τιμή στην οποία αρχικοποιήθηκε στην πρώτη γραμμή της checkContinue. Στη συνέχεια τυπώνεται η τιμή της i, δηλαδή το 1. Ακολούθως ελέγχεται η τιμή της i, η συνθήκη i==1 βρίσκεται αληθής και εκτελείται το continue. Αν το continue κατηύθυνε τη ροή στην αρχή του βρόχου, τότε το i θα γινόταν 2 και στη συνέχεια θα τυπωνόταν πριν φτάσει στη συνθήκη ελέγχου και εξέλθει από τον βρόχο. Όμως, η continue κατευθύνει άμεσα τη ροή στη συνθήκη ελέγχου η οποία φυσικά για την τιμή του i ίση με το 1 είναι ψευδής και έτσι ο βρόχος διακόπτεται έχοντας εκτυπώσει μόνο το 1.

5.2.3 Η επαναληπτική δομή for

Η for παρουσιάζει μεγάλη ευελιξία και είναι ίσως η συχνότερα χρησιμοποιούμενη επαναληπτική δομή. Η γενική μορφή της έχει ως εξής:

```
for (initialization; condition; statement) {
     statement(s);
}
```

Η for επομένως αποτελείται από τέσσερα μέρη. Το initialization είναι το τμήμα εκείνο στο οποίο τυπικά αρχικοποιείται ή και δηλώνεται η μεταβλητή ελέγχου του βρόχου. Η condition είναι η συνθήκη που ελέγχει αν θα συνεχιστεί η επαναληπτική διαδικασία ή όχι. Το statement μπορεί να είναι μια οποιαδήποτε εντολή ή συνάρτηση. Στην τυπική περίπτωση, στο statement μεταβάλλεται η τιμή της μεταβλητής ελέγχου. Το τέταρτο μέρος είναι το μπλοκ της for στο οποίο μπορούμε να τοποθετήσουμε ένα ή περισσότερα statements.

Το initialization εκτελείται μια φορά κατά την είσοδο στον βρόχο. Στη συνέχεια εκτελείται η condition. Αν βρεθεί αληθής, η ροή κατευθύνεται στο μπλοκ της for. Αφού εκτελεστούν οι statements του μπλοκ, η ροή κατευθύνεται στο statement εντός του κύριου σώματος της for. Μετά την εκτέλεση του statement εντός του κύριου σώματος, η ροή κατευθύνεται στην condition για να ακολουθήσει μια κυκλική διαδρομή του τύπου condition-statement(s)-statement έως ότου βέβαια η condition λάβει την τιμή false. Σε αυτήν την περίπτωση, διακόπτεται η for και η ροή του προγράμματος κατευθύνεται στην έξοδο του βρόχου.
Ας δούμε καταρχάς μια απλή εφαρμογή της for. Ο κώδικας 5.18, τυπώνει τους ακεραίους από 0 έως 9.

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

Κώδικας 5.18 *Εκτύπωση των ακέραιων 0..9 με την for*

Η πρώτη τιμή που εκτυπώνεται είναι η τιμή 0. Αυτό δείχνει πως η ροή εκτέλεσης πρώτα εισέρχεται στο μπλοκ της for και στη συνέχεια εκτελεί την προσαύξηση του i στο κυρίως σώμα της for. Η δε τελευταία τιμή είναι το 9. Πράγματι, μόλις τυπωθεί το 9, η ροή κατευθύνεται στην πρόταση προσαύξησης του i που γίνεται 10. Στη συνέχεια, η συνθήκη ελέγχου καθίσταται ψευδής και οδηγεί στην έξοδο του βρόχου.
Ας σημειωθεί πως στο τμήμα αρχικοποίησης της for δεν είναι υποχρεωτικό να δηλώνεται η μεταβλητή ελέγχου. Ωστόσο, αποτελεί καλή πρακτική η δήλωσή της εφόσον αυτή δεν είναι χρήσιμη έξω από την for. Κάτι ακόμη που πρέπει να αποσαφηνίσουμε είναι πως κανένα από τα τέσσερα τμήματα της for δεν είναι υποχρεωτικό.

Στη συνέχεια δίνουμε παραδείγματα χρήσης της for μέσα από τα οποία γίνονται σαφείς οι παρατηρήσεις μας σε σχέση με αυτήν. Πιο συγκεκριμένα, υλοποιούμε με for κώδικες που αξιοποιούν την while ή την do-while και έχουν ήδη παρουσιαστεί μέχρι εδώ σε αυτήν την ενότητα. Με αυτόν τον τρόπο θα αποκτήσουμε πληρέστερη εικόνα της σχετικής ευελιξίας της for.
Ο κώδικας 5.10 που επιδεικνύει τη χρήση της while, υλοποιείται με for όπως δείχνει ο κώδικας 5.19.

```
static void forDemo() { //Κώδικας 5.19
    int i;
    for (i = 10; i > 0; i--) {
        System.out.print("i>0 ");
    }
```

```
        System.out.println("i=" + i);
    }
```

Κώδικας 5.19 *Ίδιο αποτέλεσμα με την while του κώδικα 5.10*

Στον κώδικα 5.19 η μεταβλητή ελέγχου i δεν δηλώνεται στο τμήμα αρχικοποίησης της for. Αυτό είναι αναγκαίο, καθώς θέλουμε να προσπελάσουμε τη μεταβλητή έξω από τα όρια της for. Αν την είχαμε δηλώσει στο τμήμα αρχικοποίησης της for, η εμβέλεια της μεταβλητής θα περιοριζόταν στα τμήματα της for.
Η συνάρτηση prtRandoms που παρουσιάζεται στον κώδικα 5.12 μεταγράφεται με for όπως φαίνεται στον κώδικα 5.20:

```
    static void prtRandoms() { //Κώδικας 5.20
        Random generator = new Random();
        int rI;
        for (int i = 0; i++ < 10;) {
            rI = generator.nextInt(10) + 1;
            System.out.println(rI);
            if (rI < 2) {
                break;
            }
        }
    }
```

Κώδικας 5.20 *Η prtRandoms του κώδικα 5.12 υλοποιημένη με for*

Προσέξτε πως σε αυτήν την περίπτωση το τρίτο τμήμα της for δεν χρησιμοποιήθηκε. Όπως αναφέραμε προηγουμένως, όλα τα τμήματα της for είναι προαιρετικά.
Η συνάρτηση prtRandoms που παρουσιάστηκε στον κώδικα 5.13 μεταγράφεται με for όπως δείχνει ο κώδικας 5.21:

```
    static void prtRandoms() { //Κώδικας 5.21
        int rI;
        Random generator = new Random();
        for (int i = 0; i < 10; i++) {
            rI = generator.nextInt(10) + 1;
            if (rI == 3 || rI == 4) {
                continue;
            }
            System.out.println(rI);
        }
    }
```

Κώδικας 5.21 *Η prtRandoms του κώδικα 5.13 υλοποιημένη με for*

Ο κώδικας 5.22 υλοποιεί την calcSum του κώδικα 5.14 χρησιμοποιώντας for αντί για do-while:

```
    static void calcSum() { //Κωδικας 5.22
        int sum = 0;
        Scanner input = new Scanner(System.in);
        for (int i = 1; i != 0;) {
            System.out.print("Enter int (0 terminates input):");
            i = input.nextInt();
            sum += i;
        }
        System.out.println("Sum = " + sum);
    }
```

Κώδικας 5.22 *Η calcSum του κώδικα 5.14 υλοποιημένη με for*

Ο κώδικας 5.23 παρέχει μια κάπως διαφορετική υλοποίηση του κώδικα 5.18:

```
     for (int i=0; i<10; System.out.println(i));
```

Κώδικας 5.23 *Εναλλακτική υλοποίηση του κώδικα 5.18*

Συχνά οι αρχάριοι προγραμματιστές έχουν την εντύπωση πως οι μεταβλητές ελέγχου της for είναι υποχρεωτικά τύπου int. Κάτι τέτοιο βέβαια δεν αληθεύει. Παρακάτω στον κώδικα 5.24, εκτυπώνουμε το αγγλικό αλφάβητο με κεφαλαίους χαρακτήρες:

```
static void prtAlpha() { //Κώδικας 5.24
    for (char c = 'A'; c <= 'Z'; c++) {
        System.out.print(c + " ");
    }
    System.out.println();
}
```

Κώδικας 5.24 *Εκτύπωση της Αγγλικής Αλφάβητου με for*

Η μεταβλητή που αρχικοποιείται στο τμήμα αρχικοποίησης της for δεν λαμβάνει υποχρεωτικά μέρος στη συνθήκη ελέγχου. Αυτό επιδεικνύεται στον κώδικα 5.25, ο οποίος τυπώνει διαδοχικούς ακέραιους αρχίζοντας από το 0 έως ότου ο χρήστης επιλέξει έξοδο.

```
static void prtInts() {//Κώδικας 5.25
    int v = 1;
    Scanner input = new Scanner(System.in);
    for (int i = 0; v > 0; i++) {
        System.out.println(i);
        System.out.print("Εισάγετε 0 για έξοδο:");
        v = input.nextInt();
    }
}
```

Κώδικας 5.25 *Εκτύπωση διαδοχικών ακέραιων μέχρι να επιλέξει ο χρήστης έξοδο*

Όπως στις δομές επιλογής, έτσι και εδώ έχουμε τη δυνατότητα εμφωλευμένων επαναληπτικών δομών. Ο κώδικας 5.26 τυπώνει 10 φορές το αγγλικό αλφάβητο.

```
static void nestedFor() { //Κώδικας 5.26
    for (int i = 0; i < 10; i++) {
        for (char c = 'A'; c <= 'Z'; c++) {
            System.out.print(c + " ");
        }
        System.out.println();
    }
}
```

Κώδικας 5.26 *Παράδειγμα εμφωλευμένης for*

Ο κώδικας 5.27 έχει ως σκοπό να δώσει έμφαση στο γεγονός ότι τα διάφορα τμήματα της for είναι προαιρετικά. Τέτοιου είδους χρήση της for υποστηρίζεται μεν από τον μεταγλωττιστή, είναι ωστόσο ασυνήθιστη.

```
static void emptyFor() { //Κώδικας 5.27
    int i = 0;
    for (;;) {
        System.out.println(i++);
        if (i > 10) {
            break;
        }
    }
}
```

Κώδικας 5.27 *Παράδειγμα της for χωρίς κανένα από τα τρία κύρια τμήματα*

Σημείωση: Υπάρχει και μια επιπλέον μορφή της for, γνωστή ως ενισχυμένη (enhanced) for. Η ενισχυμένη for σχετίζεται με τις συλλογές, δηλαδή με δομές που είναι κατάλληλες για αποθήκευση πλήθους δεδομένων, π.χ. πίνακες. Η ενισχυμένη for παρουσιάζεται στην ενότητα 6.1.2.

5.3. Λυμένες Ασκήσεις

5.3.1 Εκτύπωση πραγματικών με δεκαδικό βήμα

Αναπτύξτε συνάρτηση που τυπώνει τους πραγματικούς αριθμούς από 1 έως 10 με βήμα 0.1.

Λύση
Μια απλή for με μεταβλητή ελέγχου τύπου double αρκεί για τη λύση αυτής της άσκησης.

```
static void simpleForwithDouble() { //Κώδικας 5.28
    for (double i = 0; i <= 10; i += 0.1) {
        System.out.println(i);
    }
}
```

Κώδικας 5.28 *Απλή for με μεταβλητή ελέγχου double και βήμα 0.1*

5.3.2 Εκτύπωση ακέραιων υπό συνθήκη

Να γράψετε πρόγραμμα που τυπώνει όλους τους ακέραιους από το 0..100 που διαιρούνται ακριβώς με το 4, το 6 και το 8.

Λύση
Η πρώτη σκέψη που μας έρχεται στο μυαλό είναι να ελέγξουμε κάθε ακέραιο από το 0 έως το 100 και αν διαιρείται ακριβώς με το 4, το 6 και το 8 να το εμφανίσουμε. Ωστόσο, υπάρχει και πιο αποδοτικός τρόπος. Αφού οι αριθμοί που θα τυπώνουμε διαιρούνται με το 4, το 6 και το 8, μπορούμε να ξεκινήσουμε από το 4 και να ελέγξουμε μόνο τα πολλαπλάσια του 4. Ελέγχουμε λοιπόν τα πολλαπλάσια του 4. Αν πληρούν και τις λοιπές προϋποθέσεις, τότε τα τυπώνουμε. Αν θεωρήσουμε πως και το 0 είναι πολλαπλάσιο του 4, τότε θα ξεκινήσουμε από το 0. Με αυτόν τον τρόπο, αντί να ελέγξουμε 100 περίπου ακέραιους, ελέγχουμε μόνο 25. Μπορούμε και εδώ να χρησιμοποιήσουμε μια for.

```
static void prtRealsConditional() { //Κώδικας 5.29
    for (int i = 0; i <= 100; i += 4) {
        if (i % 6 == 0 && i % 8 == 0) {
            System.out.println(i);
        }
    }
}
```

Κώδικας 5.29 *Εκτύπωση ακέραιων υπό συνθήκη*

5.3.3 Εκτύπωση και καταμέτρηση ακέραιων υπό συνθήκη

Να γράψετε πρόγραμμα που τυπώνει όσους ακεραίους στην περιοχή 0..1000 είναι πολλαπλάσια του 3 και ταυτοχρόνως τέλεια τετράγωνα, δηλαδή η τετραγωνική τους ρίζα είναι ακέραιος.
Μετράει και εκτυπώνει το πλήθος των ακέραιων που πληρούν τις περιγραφόμενες προϋποθέσεις.
Σημείωση: Η τετραγωνική ρίζα μιας αριθμητικής μεταβλητής i δίνεται από τη συνάρτηση sqrt της κλάσης Math : Math.sqrt(i).

Λύση
Το ερώτημα εδώ είναι ο έλεγχος για να διαπιστώσουμε αν η τετραγωνική ρίζα ενός αριθμού είναι ακέραιος. Καταρχάς την τετραγωνική ρίζα θα την υπολογίσουμε με τη συνάρτηση sqrt() της Math. Όπως είναι λογικό, η

συνάρτηση αυτή επιστρέφει αριθμό τύπου double. Για να διαπιστώσουμε αν ο αριθμός αυτός είναι ακέραιος, θα ελέγξουμε αν το υπόλοιπο της διαίρεσης μεταξύ τετραγωνικής ρίζας και 1 είναι ίσο με το 0. Προκειμένου όμως να ελέγξουμε σωστά θα πρέπει να χρησιμοποιήσουμε προσεγγιστική ισότητα, όπως έχουμε επισημάνει στις ενότητες 3.2 και 4.1.4.1.

```
    static boolean approximateEquals(double f1, double f2, double epsilon) {
        return Math.abs(f2 - f1) < epsilon;
    }

    static void search() { //Κώδικας 5.30
        int cnt = 0;
        for (int i = 0; i < 1000; i += 3) {
            if (approximateEquals(Math.sqrt(i) % 1, 0, 0.00001d)) {
                System.out.print(i + " ");
                cnt++;
            }
        }
        System.out.println();
        System.out.println("Ta teleia tetragwna twn pollaplasiwn tou 3 apo
0..1000 einai " + cnt);
    }
```

Κώδικας 5.30 *Αναζήτηση σε πίνακα πραγματικών με προσεγγιστική ισότητα*

5.3.4 Τυχαίοι ακέραιοι και switch

Να γράψετε πρόγραμμα που λαμβάνει 2 ακέραιους από τον χρήστη και ανάλογα με τις τιμές τους, τυπώνει ένα μήνυμα της μορφής "εποχή ημέρα". Η ονομασία ημέρας και εποχής με βάση τις ακόλουθες αντιστοιχίσεις: (1->Κυριακή, 2->Δευτέρα, …, 7 Σάββατο), (1-> Άνοιξη, 2-> Καλοκαίρι, 3-> Φθινόπωρο, 4->Χειμώνας). Σε περίπτωση που η τιμή ενός (ή και των δύο) ακέραιων είναι εκτός των προαναφερόμενων περιοχών τυπώνει unkown value.

Λύση
Εδώ είναι βολικό να υπολογίσουμε πρώτα το μήνυμα και μετά να το τυπώσουμε. Για τον σκοπό αυτό θα χρησιμοποιήσουμε κατάλληλες μεταβλητές τύπου String.

```
    static void decodeDaySeason() { //Κώδικας 5.31
        Scanner in = new Scanner(System.in);
        System.out.print("Εισάγετε ΑΑ ημέρας: ");
        int day = in.nextInt();
        System.out.print("Εισάγετε ΑΑ εποχής: ");
        int season = in.nextInt();
        String message = "";
        if (day < 1 || day > 7 || season < 1 || season > 4) {
            message = "Unkown Value";
        } else {
            switch (day) {
                case 1:
                    message = "Sunday";
                    break;
                case 2:
                    message = "Monday";
                    break;
                case 3:
                    message = "Tuesday";
                    break;
                case 4:
                    message = "Wensday";
                    break;
                case 5:
```

```
                    message = "Thursday";
                    break;
                case 6:
                    message = "Friday";
                    break;
                case 7:
                    message = "Saturday";
                    break;
            }
            switch (season) {
                case 1:
                    message += " Spring";
                    break;
                case 2:
                    message += " Summer";
                    break;
                case 3:
                    message += " Autumn";
                    break;
                case 4:
                    message += " Winter";
                    break;
            }
        }
        System.out.println(message);
    }
```

Κώδικας 5.31 *Εκτύπωση ημέρας και εποχής*

Στον κώδικα 5.31, καταρχάς, εισάγουμε από το πληκτρολόγιο δύο ακέραιους, ο ένας αντιπροσωπεύει τον αύξοντα αριθμό της ημέρας και ο άλλος της εποχής. Στη συνέχεια δηλώνουμε και αρχικοποιούμε τη μεταβλητή message τύπου String. Η αρχικοποίηση στη συγκεκριμένη περίπτωση είναι υποχρεωτική. Αν δεν αρχικοποιήσουμε την message, ο μεταγλωττιστής θα παράγει σφάλμα σε όλα τα σημεία του κώδικα που επιχειρείται η σύνδεση της message με την περιγραφή της εποχής. Πριν μπούμε στον έλεγχο της ημέρας στην πρώτη switch, ελέγχουμε αν οι αύξοντες αριθμοί είναι στα επιτρεπτά όρια. Αν κάποιος από τους ακέραιους που θα εισάγει ο χρήστης είναι εκτός ορίων, τότε το message έχει λάβει τιμή. Επιπλέον, η ροή του κώδικα δεν θα περάσει από τις επίμαχες γραμμές. Αν πάλι κανένας από τους ακέραιους δεν είναι εκτός ορίων, το message θα έχει λάβει τιμή από την πρώτη switch. Επομένως, το message θα έχει λάβει οπωσδήποτε τιμή πριν η ροή κατευθυνθεί στη δεύτερη switch. Ωστόσο, το αποτέλεσμα δεν μπορεί να διαπιστωθεί κατά τον χρόνο μεταγλώττισης. Για τον μεταγλωττιστή είναι πιθανό η ροή να μπει στη δεύτερη switch χωρίς να έχει αρχικοποιηθεί το message. Για παράδειγμα, αν δεν ελέγξουμε την day για την τιμή 1, αλλά την ελέγξουμε για μια άλλη τιμή, π.χ. 9, τότε το message μπορεί να φτάσει στη δεύτερη switch αναρχικοποίητο.

5.3.5 Έλεγχος αν ο ακέραιος είναι πρώτος ή όχι

Να γράψετε πρόγραμμα που λαμβάνει έναν ακέραιο από τον χρήστη και εξετάζει αν είναι πρώτος αριθμός ή όχι.
Σημείωση: Πρώτος είναι ένας ακέραιος μεγαλύτερος της μονάδας που διαιρείται ακριβώς μόνο με το 1 και τον εαυτό του.

Λύση
Όλοι οι ακέραιοι, πρώτοι και μη, διαιρούνται ακριβώς με το 1 και τον εαυτό τους. Αν όμως ένας ακέραιος διαιρείται ακριβώς με οποιονδήποτε άλλον πλην του 1 και του εαυτού του, τότε δεν είναι πρώτος. Επομένως θα πρέπει να εξετάσουμε αν ο ακέραιος που θα εισάγει ο χρήστης διαιρείται ακριβώς με οποιονδήποτε άλλον ακέραιο. Αν θεωρήσουμε έναν ακέραιο x, ποιοι είναι οι ακέραιοι εκτός του 1 και του x που έχουν πιθανότητα να διαιρούν ακριβώς τον x; Είναι προφανές πως οι μεγαλύτεροι του x δεν έχουν τέτοια πιθανότητα. Επίσης αποκλείονται το 1 και το x. Επομένως μένουν οι ακέραιοι από το 2 έως x-1. Πράγματι αν εξετάσουμε αν το x διαιρείται ακριβώς με οποιονδήποτε ακέραιο από 2 έως x-1, μπορούμε να διαπιστώσουμε αν ο x είναι πρώτος

ή όχι. Αν όμως σκεφτούμε λίγο πιο προσεκτικά, θα αντιληφθούμε πως οι αριθμοί που είναι μεγαλύτεροι από x/2 επίσης αποκλείεται να διαιρούν ακριβώς τον x καθώς ο μεγαλύτερος πιθανός διαιρέτης του είναι ο x/2. Συνεπώς αρκεί να εξετάσουμε τους ακέραιους από 2 έως και x/2.

```
static void prwtos() { //Κώδικας 5.32
    Scanner in = new Scanner(System.in);
    System.out.print("Εισάγετε ακέραιο: ");
    int k = in.nextInt();

    boolean isPrwtos = k >= 2;
    if (isPrwtos) {
        for (int i = 2; i <= k / 2; i++) {
            if (k % i == 0) {
                isPrwtos = false;
                break;
            }
        }
    }
    if (isPrwtos) {
        System.out.println(k + " είναι πρώτος");
    } else {
        System.out.println(k + " δεν είναι πρώτος");
    }
}
```

Κώδικας 5.32 *Έλεγχος αν ένας ακέραιος είναι πρώτος αριθμός*

Καταρχάς λαμβάνουμε την είσοδο του χρήστη στην ακέραια μεταβλητή k. Στη συνέχεια, δηλώνουμε και αρχικοποιούμε τη μεταβλητή isPrwtos τύπου boolean. Η isPrwtos αρχικοποιείται στην τιμή της έκφρασης k>=2, δηλαδή στην τιμή true αν το k είναι μεγαλύτερο ή ίσο του 2 και στην τιμή false αν το k είναι μικρότερο από το 2. Στη δεύτερη αυτή περίπτωση, το k εξ ορισμού δεν είναι πρώτος αριθμός. Επομένως ο κώδικας δεν θα μπει καθόλου στην if που ακολουθεί αλλά θα μεταβεί κατευθείαν στην else της επόμενης if και θα τυπώσει πως ο αριθμός δεν είναι πρώτος.

Αντίθετα, αν το k είναι μεγαλύτερο ή ίσο με το 2, η ροή του κώδικα θα εισέλθει στην πρώτη if. Στη συνέχεια ο έλεγχος περνάει στην επαναληπτική διαδικασία που ακολουθεί. Ο σκοπός αυτού του βρόχου είναι να ελέγξει αν το k διαιρείται ακριβώς με οποιονδήποτε ακέραιο από το 2 έως το k/2. Αν βρεθεί έστω ένας ακέραιος που διαιρεί ακριβώς το k, τότε το k δεν είναι πρώτος οπότε η isPrwtos ενημερώνεται με την τιμή false και η επαναληπτική διαδικασία διακόπτεται. Στη συνέχεια η ροή κατευθύνεται στη δεύτερη if η οποία ανάλογα με την τιμή της isPrwtos τυπώνει το κατάλληλο μήνυμα.

5.3.6 Εκτύπωση Αγγλικής αλφάβητου

Να γράψετε πρόγραμμα που τυπώνει 100 φορές την Αγγλική αλφάβητο με πεζούς χαρακτήρες αντεστραμμένη, δηλαδή από το z στο a. Δώστε 2 λύσεις. Η πρώτη να χρησιμοποιεί μόνο for και η δεύτερη να χρησιμοποιεί συνδυασμό while και do-while.

Λύση
Ακολουθούν οι ζητούμενες υλοποιήσεις:

```
static void printReversedAlphabet() { //Κώδικας 5.33
    for (int I = 0; i < 100; i++) {
        for (char c ='''; c >='''; c--) {
            System.out.print(c +"""");
        }
        System.out.println();
    }
}

static void printReversedAlphabet2() { //Κώδικας 5.33
```

```
        int i = 0;
        while (i++ < 100) {
            char c ='''';
            do {
                System.out.print(c +""");
                c--;
            } while (c >='''');
            System.out.println();
        }
    }
```

Κώδικας 5.33 *Εκτύπωσηιςς Αγγλικής αλφαβήτου με αντεστραμμένη διάταξη. Δύο προσεγγίσεις.*

5.3.7 Εκθετική εξίσωση

Γράψτε κατάλληλο κώδικα που υπολογίτις τις τιμές των a,b,c στην 2a*3b*5c=22500, με a<20, b<20 και c<20. Σε περίπτωση που δεν βρεθεί λύση, το πρόγραμμα να εμφανίζει το μήνυμα “Δεν βρέθηκε λύση”.

Λύση
Ο πιο άμεσος τρόπος για τη λύση αυτής της άσκησης είναι η χρήση εμφωλευμένων for βάθους τριών επιπέδων. Θα πρέπει όμως να προσέξουμε ώστε να χρησιμοποιήσουμε έλεγχο προσεγγιστικής ισότητας για τη σύγκριση πραγματικών αριθμών.

```
    static void ekthetikiExiswsi() { //Κώδικας 5.34
        boolean solutionFound = false;

        for (int a = 0; a < 20; a++) {
            for (int b = 0; b < 20; b++) {
                for (int c = 0; c < 20; c++) {
                    double rslt = Math.pow(2, a) * Math.pow(3, b) * Math.pow(5,
c);
                    if (approximateEquals(rslt, 22500, 0.000001)) {
                        solutionFound = true;
                        System.out.println("a=" + a + ", b=" + b + ", c=" + c);
                        break;
                    }
                }
            }
        }
        if (!solutionFound) {
            System.out.println("Δεν βρέθηκε λύση");
        }
    }
```

Κώδικας 5.34 *Λύση εξίσωσης*

Μία σημαντική παρατήρηση εδώ είναι πως η break μεταφέρει τη ροή του προγράμματος έξω από τον εξωτερικό βρόχο και όχι απλά έξω από τον εσωτερικό βρόχο.

5.4 Ασκήσεις προς λύση

1. Αναπτύξτε πρόγραμμα που δέχεται έναν ακέραιο ως είσοδο από τον χρήστη και τυπώνει όλους τους διαιρέτες του.

2. Αναπτύξτε πρόγραμμα που λαμβάνει δύο ακεραίους από τον χρήστη και τυπώνει τον Μέγιστο Κοινό Διαιρέτη τους.

3. Αναπτύξτε πρόγραμμα που λαμβάνει έναν ακέραιο x από τον χρήστη και τυπώνει όλους τους πρώτους αριθμούς που περιλαμβάνονται μεταξύ του 2 και του x.

4. Αναπτύξτε πρόγραμμα που παράγει ερωτήσεις πρόσθεσης ή αφαίρεσης με τυχαίους τελεστέους, ζητά τις απαντήσεις από τον χρήστη και παρέχει αξιολόγηση των απαντήσεων στη μορφή:

```
Απάντησες σωστά σε k από m ερωτήσεις πρόσθεσης
Απάντησες σωστά σε r από t ερωτήσεις αφαίρεσης
Μέσος όρος βαθμολογίας : y
```

5. Αναπτύξτε πρόγραμμα που ζητά από τον χρήστη των αριθμό ωρών που εργάστηκε κάθε ημέρα μιας εβδομάδας, το ωρομίσθιό του και υπολογίζει:

1. Ποια ημέρα εργάστηκε τις περισσότερες ώρες;
2. Ποια ημέρα εργάστηκε τις λιγότερες ώρες;
3. Πόσες ώρες κατά μέσο όρο εργάστηκε την κάθε ημέρα της εβδομάδας;
4. Ποια είναι η αμοιβή της εβδομάδας σύμφωνα με τις ώρες που εργάστηκε και το ωρομίσθιό του;

6. Αναπτύξτε πρόγραμμα που λαμβάνει ένα αριθμό από τον χρήστη, έστω n, και υπολογίζει τον μικρότερο ακέραιο k για τον οποίο ισχύει $k^2>n$

7. Αναπτύξτε πρόγραμμα που λαμβάνει είσοδο από τον χρήστη μια ακέραια τιμή που αναπαριστά ένα έτος και υπολογίζει αν το έτος είναι δίσεκτο. Βρείτε τον ορισμό του δίσεκτου έτους στο διαδίκτυο.

8. Ένας πωλητής λαμβάνει μηνιαίο μισθό 1200 Ευρώ. Επιπλέον, του μισθού λαμβάνει προμήθεια επί των πωλήσεων. Για πωλήσεις από 0 έως 10.000 Ευρώ, η προμήθεια είναι 5%. Για πωλήσεις από 10.000 έως 20.000 Ευρώ, η προμήθεια είναι 10%. Για πωλήσεις από 20.000 Ευρώ και πάνω, η προμήθεια είναι 20%. Επομένως, ο πωλητής για έναν μήνα που πραγματοποίησε πωλήσεις 15.000 Ευρώ θα λάβει προμήθεια ίση με 10.000x5%+5.000x10%. Αναπτύξτε πρόγραμμα που ζητά από τον πωλητή τις πωλήσεις για κάθε μήνα ενός έτους και υπολογίζει και εκτυπώνει τόσο την μηνιαία αμοιβή του όσο και την ετήσια.

9. Η εταιρία Service απασχολεί τρεις κατηγορίες εργαζόμενων, κατόχους απολυτηρίου Λυκείου, κατόχους Πανεπιστημιακού πτυχίου και κατόχους Διδακτορικού τίτλου. Τα ωρομίσθια των εργαζόμενων έχουν ως εξής: 40 Ευρώ για τους κατόχους απολυτηρίου Λυκείου, 60 Ευρώ για τους κατόχους πανεπιστημιακού πτυχίου και 80 Ευρώ για τους κατόχους Διδακτορικού τίτλου. Το τυπικό ωράριο για κάθε εργαζόμενο είναι 40 ώρες την εβδομάδα. Κάθε επιπλέον ώρα εργασίας αμείβεται με προσαύξηση 50% επί του ωρομισθίου του εργαζόμενου. Γράψτε πρόγραμμα που ζητά από τον εργαζόμενο την κατηγορία στην οποία ανήκει, τον αριθμό ωρών που εργάστηκε την προηγούμενη εβδομάδα και υπολογίζει πόσες είναι οι τακτικές αποδοχές της εβδομάδας, πόσες οι αποδοχές λόγω υπερωρίας και ποιο το σύνολο των αποδοχών.

10. Αναπτύξτε συνάρτηση που λαμβάνει έναν ακέραιο από τον χρήστη τον οποίο εκχωρεί σε κατάλληλη μεταβλητή αφού διασφαλίσει πως η εισαγόμενη τιμή βρίσκεται στα όρια 1 έως 10.

11. Μελετήστε τις συναρτήσεις

```
static void chkBoolean() {
     Random r = new Random();
     boolean b = r.nextBoolean();
     if (b == true) {
          System.out.println("Επιτυχία!");
     } else {
          System.out.println("Αποτυχία");
     }
}

static void chkBoolean2() {
     Random r = new Random();
     boolean b = r.nextBoolean();
     if (b) {
```

```
            System.out.println("Επιτυχία!");
        } else {
            System.out.println("Αποτυχία");
        }
    }
```

Έχουν κάποια διαφορά ως προς το αποτέλεσμα; Ποια θα προτιμούσατε και γιατί;

12. Μελετήστε τον ακόλουθο κώδικα

```
    static void bugAlphabet() { Άσκηση 5.20
        for (char c = 'z'; c >= 'a'; c--) {
            System.out.print(c + ' ');
        }
    }
```

Ποια είναι η έξοδός του; Τρέξτε τον, να διαπιστώσετε αν προβλέψατε τη σωστή έξοδο. Αν όχι, εξηγήστε ποια η διαφορά και γιατί;

14. Να μεταγραφεί η συνάρτηση

```
    public static void xEx1() {
        for (int i = 0, j = 1; i < 10 || i < j; i++) {
            j += i;
            if (i % 3 == 0) {
                j = 0;
            }
            System.out.print("(" + i + ", " + j + ") ");
        }
        System.out.println();
    }
```

Με αντικατάσταση του for με while

Με αντικατάσταση του for με do while

15. Να μεταγραφεί η συνάρτηση

```
    public static void xEx2() {
        int k = 3;
        if (k == 3) {
            System.out.print(3 + " ");
        }
        if (k == 4 || k == 3) {
            System.out.print(4 + " ");
        } else if (k == 5) {
            System.out.print(5 + " ");
        } else {
            System.out.print("unkown");
        }
        System.out.println();
    }
```

με χρήση της switch.

Κεφάλαιο 6

Σύνοψη
Στο κεφάλαιο αυτό παρουσιάζεται μια εισαγωγή στους πίνακες. Πιο συγκεκριμένα, περιλαμβάνονται η δημιουργία μονοδιάστατων και πολυδιάστατων πινάκων, η ενισχυμένη for, η δημιουργία αντιγράφων πινάκων, η ταξινόμηση, η σειριακή και δυαδική αναζήτηση, ο έλεγχος ισότητας, η εμφάνιση πίνακα, ο υπολογισμός αθροίσματος των στοιχείων ενός πίνακα, η εύρεση του μεγαλύτερου και του μικρότερου στοιχείου, η αρχικοποίηση με τυχαίες τιμές. Το κεφάλαιο περιέχει λυμένες ασκήσεις και ασκήσεις προς λύση.

Προαπαιτούμενη γνώση
Οι θεμελιώδεις τύποι και η βασική γνώση του τύπου String. Βασική είσοδος και έξοδος. Η λειτουργία των τελεστών. Οι δομές επιλογής και οι επαναληπτικές δομές.

Λέξεις κλειδιά
Μονοδιάστατοι πίνακες, πολυδιάστατοι πίνακες, δείκτης, ταξινόμηση, σειριακή αναζήτηση, δυαδική αναζήτηση.

6 Πίνακες

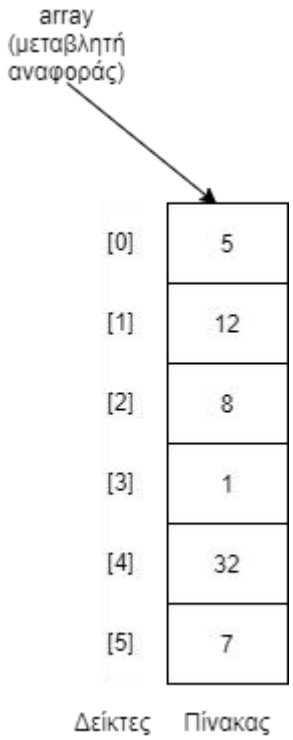
Οι πίνακες είναι συλλογές δεδομένων ίδιου τύπου. Για παράδειγμα, ένας πίνακας ακέραιων αποθηκεύει μια σειρά από τιμές τύπου int ή ένας πίνακας πραγματικών αποθηκεύει μια σειρά από τιμές τύπου double. Όπως εξηγούμε αναλυτικά παρακάτω σε αυτήν την ενότητα, η προσπέλαση στα στοιχεία ενός πίνακα είναι η ταχύτερη δυνατή σε σχέση με την προσπέλαση στα στοιχεία συλλογής οποιουδήποτε άλλου τύπου.

Στο πλαίσιο της Java, ένας πίνακας τα στοιχεία του οποίου δεν είναι πίνακες ονομάζεται μονοδιάστατος πίνακας. Ένας πίνακας τα στοιχεία του οποίου είναι μονοδιάστατοι πίνακες ονομάζεται δισδιάστατος πίνακας. Αντίστοιχα, ένας πίνακας τα στοιχεία του οποίου είναι δισδιάστατοι πίνακες ονομάζεται τρισδιάστατος πίνακας, κόκ. Συχνά οι μη μονοδιάστατοι πίνακες αναφέρονται ως πολυδιάστατοι.

Σε αυτό το κεφάλαιο, παρουσιάζουμε τις συντακτικές λεπτομέρειες ορισμού των μονοδιάστατων και πολυδιάστατων πινάκων, εξηγούμε βασικά στοιχεία της ενσωματωμένης στην Java, κλάσης Arrays που παρέχει μια σειρά από συναρτήσεις για τη διαχείριση πινάκων και παρουσιάζουμε μια σειρά από βασικές επεξεργασίες που αφορούν τους πίνακες.

6.1 Μονοδιάστατοι Πίνακες

Για τη διαχείριση ενός πίνακα, χρειάζεται στο πρόγραμμά μας να έχουμε τουλάχιστον μια μεταβλητή μέσω της οποίας μπορούμε να προσπελάσουμε τον πίνακα και τα στοιχεία του. Στην Java, οι μεταβλητές μέσα από τις οποίες προσπελαύνουμε πίνακες είναι μεταβλητές αναφοράς και όχι τιμής, δηλαδή αποθηκεύουν τη διεύθυνση μνήμης στην οποία είναι κατανεμημένος ο πίνακας. Επιπλέον, τα στοιχεία ενός μονοδιάστατου πίνακα κατανέμονται σε διαδοχικές θέσεις στη μνήμη. Σε αυτό το χαρακτηριστικό οφείλεται η ταχύτατη προσπέλαση των στοιχείων ενός πίνακα.
Στο σχήμα 6.1 παρουσιάζουμε έναν πίνακα ακέραιων.



Σχήμα 6.1 *Αναπαράσταση μονοδιάστατου πίνακα ακέραιων*

Ο πίνακας του σχήματος 6.1 έχει μήκος 6 και περιέχει τους ακέραιους: 5, 12, 8, 1, 32, 7. Η μεταβλητή αναφοράς array περιέχει τη διεύθυνση της αρχής του πίνακα. Οι θέσεις του πίνακα είναι αριθμημένες. Στη θέση 0, για παράδειγμα, είναι τοποθετημένη η τιμή 5, στη θέση 1 είναι τοποθετημένη η τιμή 12, κοκ.

Για να προσπελάσουμε ένα στοιχείο του πίνακα χρειαζόμαστε τρία στοιχεία. Τη μεταβλητή αναφοράς που δείχνει στην αρχή του πίνακα, έναν δείκτη (index) που δείχνει τη θέση μέσα στον πίνακα του στοιχείου που θέλουμε να προσπελάσουμε και τον τελεστή θέσης (index operator) που συμβολίζεται ως []. Σύμφωνα με τα παραπάνω, η έκφραση array[2] προσπελαύνει τη θέση με δείκτη 2 του πίνακα array όπου είναι αποθηκευμένη η ακέραια τιμή 8. Αντίστοιχα, array[3]==1, array[4]==32, κλπ.

Οι δείκτες των πινάκων στην Java αριθμούνται πάντα αρχίζοντας από το μηδέν (zero based). Έτσι, ένας πίνακας με μήκος ν έχει δείκτες από 0 έως ν-1. Ο πίνακας του σχήματος 6.1 για παράδειγμα, έχει μήκος 6 και δείκτες από 0 έως 5. Επομένως, αν επιχειρήσουμε να προσπελάσουμε το στοιχείο array[6] επιχειρούμε στην ουσία να προσπελάσουμε περιοχή έξω από τη μνήμη που έχει δεσμευτεί για τον πίνακα. Τέτοια ενέργεια συνιστά λάθος χρόνου εκτέλεσης και παράγει κατάλληλη εξαίρεση.

Όπως αναφέραμε στην αρχή αυτής της ενότητας, οι πίνακες είναι οι ταχύτερες συλλογές. Πιο συγκεκριμένα, οι πίνακες είναι δομές τυχαίας προσπέλασης (random access), δηλαδή δομές στις οποίες ο χρόνος που απαιτείται για την προσπέλαση οποιουδήποτε στοιχείου είναι σταθερός. Με άλλα λόγια, χρειαζόμαστε τον ίδιο χρόνο για να προσπελάσουμε ένα στοιχείο στη θέση 1000 ενός πίνακα με τον χρόνο που απαιτεί η προσπέλαση στη θέση 0 του πίνακα.

Στον πίνακα του σχήματος 6.1, το στοιχείο στη θέση 0 βρίσκεται στη διεύθυνση μνήμης array+0*Integer.SIZE και το στοιχείο στη θέση 1 βρίσκεται στη διεύθυνση μνήμης array+1*Integer.SIZE. Γενικότερα, το στοιχείο στη θέση k ενός μονοδιάστατου πίνακα ακέραιων βρίσκεται στη διεύθυνση αρχής του πίνακα + k*Integer.SIZE. Επομένως, για να υπολογιστεί η διεύθυνση ενός οποιουδήποτε στοιχείου του πίνακα, χρειαζόμαστε να υπολογίσουμε μόνο μια απλή έκφραση.

Αυτή η ευκολία είναι αποτέλεσμα της διαδοχικής κατανομής στη μνήμη των στοιχείων των μονοδιάστατων πινάκων. Το μειονέκτημα βέβαια της διαδοχικής κατανομής είναι πως μετά τη δημιουργία δεν είναι δυνατόν να μεταβληθεί το μέγεθος του πίνακα. Ωστόσο, αυτό το μειονέκτημα, εύκολα μπορούμε να το παρακάμψουμε όπως θα δούμε παρακάτω στην άσκηση 6.5.7.

6.1.1 Δημιουργία μονοδιάστατων πινάκων

Σε αυτήν την ενότητα παρουσιάζονται οι τρόποι βασικής διαχείρισης των μονοδιάστατων πινάκων. Πιο συγκεκριμένα, θα δούμε πώς δηλώνουμε έναν μονοδιάστατο πίνακα, πώς τον αρχικοποιούμε και πώς προσπελαύνουμε τα στοιχεία του.
Στον κώδικα 6.1, δηλώνουμε, δημιουργούμε και αρχικοποιούμε τα στοιχεία ενός μονοδιάστατου πίνακα ακέραιων.

```
1   int[] array;
2   array = new int[3];
3   array[0] = 1;
4   array[1] = 3;
5   array[2] = 5;
```

Κώδικας 6.1 *Δήλωση, δημιουργία και αρχικοποίηση μονοδιάστατου πίνακα ακέραιων*

Στη γραμμή 1 του κώδικα 6.1 δηλώνουμε τη μεταβλητή array που ο τύπος της είναι πίνακας ακέραιων. Σε αυτήν τη φάση δεν έχει δημιουργηθεί ακόμη ο πίνακας. Έχει απλά δηλωθεί μια μεταβλητή κατάλληλη για τη διαχείριση μονοδιάστατων πινάκων ακέραιων. Αν θέλαμε μονοδιάστατους πίνακες πραγματικών αριθμών, θα έπρεπε να δηλώσουμε double[] array. Γενικότερα, μπορούμε να δηλώσουμε πίνακες οποιουδήποτε τύπου στην Java.

Στη γραμμή 2, δημιουργείται ο πίνακας. Η δημιουργία του επιτυγχάνεται με τη χρήση της λέξης-κλειδί new που συνοδεύεται από τον αριθμό 3 μέσα σε τετράγωνες παρενθέσεις (square brackets). Επομένως, η εντολή new int[3], δεσμεύει στη μνήμη τρεις διαδοχικές θέσεις που στην κάθε μία μπορεί να αποθηκευτεί ένας ακέραιος. Επιπλέον, στη συγκεκριμένη περίπτωση η new επιστρέφει μεταβλητή τύπου int[], την οποία εκχωρούμε στη μεταβλητή array που δηλώσαμε στη γραμμή 1. Στις γραμμές 3 έως 5, τοποθετούμε στις θέσεις 0, 1 και 2 τις τιμές 1, 3, 5, αντίστοιχα.
Στη συνέχεια, μπορούμε να προσπελάσουμε ένα προς ένα τα στοιχεία του πίνακα χρησιμοποιώντας μια απλή for, όπως δείχνει ο κώδικας 6.2.

```
for (int i = 0; i < array.length; i++) {
    System.out.println("Η τιμή στην θέση " + i + " είναι " + array[i]);
}
```

Κώδικας 6.2 *Εμφάνιση των στοιχείων μονοδιάστατου πίνακα ακέραιων*

Όπως φαίνεται στον κώδικα 6.2, προσπελαύνουμε τα στοιχεία του πίνακα στις θέσεις 0 έως array.length-1. Αυτό καταρχάς σημαίνει ότι η δομή τύπου int[] που επιστρέφει η new περιλαμβάνει εκτός από τη διεύθυνση μνήμης στην οποία κατανεμήθηκε ο πίνακας και το μήκος του πίνακα. Παρότι ο συγκεκριμένος πίνακας είναι μήκους 3, συνίσταται η χρήση της length, καθώς όπως θα δούμε στην ενότητα 7 είναι δυνατόν ένας πίνακας να περάσει παραμετρικά σε έναν κώδικα και η χρήση σταθεράς να μας οδηγήσει σε προβλήματα.
Αντίστοιχα με τους άλλους τύπους, έτσι και για τους πίνακες, δήλωση και δημιουργία μπορούν να γίνουν στην ίδια γραμμή όπως δείχνει ο κώδικας:

```
int[] array=new int[3];
```

Επίσης, παρέχεται η δυνατότητα, δήλωση, δημιουργία και αρχικοποίηση να πραγματοποιηθούν στην ίδια γραμμή, όπως φαίνεται στον κώδικα που ακολουθεί:

```
int[] array=new int[]{1,3,5,7};
```

Τέλος, δήλωση, δημιουργία και αρχικοποίηση να πραγματοποιηθούν στην ίδια γραμμή με τη βοήθεια ενός αρχικοποιητή (initializer). Ο κώδικας που ακολουθεί δημιουργεί έναν πίνακα πραγματικών με στοιχεία 2.5, 4.0, 6.8.

```
double[] table={2.5, 4.0, 6.8};
```

Στο παράδειγμα αυτό ο αρχικοποιητής είναι η παράσταση {2.5, 4.0, 6.8}.

6.1.2 Ενισχυμένη for

Όπως αναφέραμε στην ενότητα 5.2.3 περί του βρόχου for, υποστηρίζεται μια διευρυμένη μορφή της for που σχετίζεται με συλλογές όπως οι πίνακες.
Η γενική μορφή της ενισχυμένης (enhanced) for έχει ως εξής:

```
for (dataType item : collection) (
     statement(s)
}
```

Επομένως, στην ενισχυμένη for, ορίζουμε καταρχάς μια μεταβλητή, τοποθετούμε άνω και κάτω τελεία και στη συνέχεια μια μεταβλητή που αντιπροσωπεύει μια συλλογή, π.χ. έναν πίνακα. Για παράδειγμα:

```
for (double currentItem : table) {
     System.out.println(currentItem+" ");
}
```

Τον κώδικα αυτόν μπορούμε να τον διαβάσουμε ως εξής: Για κάθε ένα στοιχείο του πίνακα table που είναι τύπου double και ονομάζουμε currentItem εκτέλεσε τις προτάσεις μέσα στο μπλοκ που ακολουθεί. Σε κάθε επαναληπτικό βήμα, μέσα στο μπλοκ έχουμε πρόσβαση στο τρέχον στοιχείο του table. Εκείνο που πρέπει να προσέξουμε είναι ότι η πρόσβαση που έχουμε στα στοιχεία του table με αυτήν την for είναι μόνο για διάβασμα (read only). Με άλλα λόγια, η μεταβλητή currentItem είναι τοπική μεταβλητή της for. Σε κάθε επαναληπτικό βήμα, αντιγράφεται η τιμή ενός στοιχείου του table στην currentItem. Επομένως, μέσα στο μπλοκ δεν προσπελαύνουμε άμεσα το στοιχείο του πίνακα, αλλά μόνο ένα αντίγραφό του.
Με τη βοήθεια των παραδειγμάτων που ακολουθούν διευκρινίζουμε αυτό το ζήτημα.

```
static void alterArray() { //Κώδικας 6.3
    int[] array = {1, 3, 5};
    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
        array[i] += 1;
    }
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i] + " ");
    }
}
```

Κώδικας 6.3 *Μεταβολή των στοιχείων του array*

Στον κώδικα 6.3 αρχικοποιούμε καταρχάς το array. Στη συνέχεια, σε κάθε επαναληπτικό βήμα μιας κλασικής for, εμφανίζουμε ένα στοιχείο του array και αμέσως μετά προσαυξάνουμε το στοιχείο κατά 1. Έξω από την πρώτη επαναληπτική διαδικασία αναπτύσσεται μια δεύτερη που σκοπό έχει να εμφανίσει τα στοιχεία του array. Πράγματι την πρώτη φορά, εμφανίζονται οι αρχικές τιμές 1, 3, 5 και στη δεύτερη for οι τιμές αυξημένες κατά 1, δηλαδή οι τιμές 2, 4, 6.

```
static void noAlteration() { //Κώδικας 6.4
    int[] array = {1, 3, 5};
    for (int cElement : array) {
        System.out.print(cElement + " ");
        cElement += 1;
    }
    System.out.println();
    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
    System.out.println();
}
```

Κώδικας 6.4 *Μεταβολή της τοπικής μεταβλητής cElement*

Αντίθετα, ο κώδικας 6.4, εμφανίζει κατά την πρώτη for τις τιμές 1, 3, 5 αλλά εμφανίζει τις ίδιες τιμές και κατά τη δεύτερη for. Αυτό συμβαίνει γιατί όπως εξηγήσαμε, η cElement δεν είναι παρά ένα αντίγραφο του τρέχοντος στοιχείου του array. Επομένως, η προσαύξηση κατά 1 που κάνουμε μέσα στην πρώτη for αφορά το αντίγραφο και όχι το στοιχείο του array. Όπως φαίνεται από την έξοδο του κώδικα, τα στοιχεία του πίνακα παραμένουν αμετάβλητα.

6.1.3 Δημιουργία αντιγράφων

Θα συζητήσουμε τώρα πώς μπορούμε να παράγουμε αντίγραφα ενός πίνακα ή ενός τμήματος πίνακα. Καταρχάς θα πρέπει να αποσαφηνίσουμε τις επιπτώσεις που προκύπτουν από το γεγονός ότι οι μεταβλητές με τις οποίες διαχειριζόμαστε τους πίνακες είναι μεταβλητές αναφοράς.
Ας δούμε λίγο τον κώδικα 6.5:

```
1     int[] array = {1, 3, 5};
2     int[] table;
3     table = array;
4     array[1] = 9;
5     System.out.println(table[1]);
```

Κώδικας 6.5 *Εκχώρηση μεταξύ πινάκων*

Στον κώδικα 6.5 δημιουργούμε καταρχάς τον πίνακα ακέραιων array. Στη συνέχεια δηλώνουμε έναν πίνακα επίσης ακέραιων που ονομάζουμε table. Στην ουσία, η table είναι μια μεταβλητή αναφοράς τύπου int[]. Στο επόμενο βήμα εκχωρούμε την array στην table. Εφόσον η array είναι μία δομή που διατηρεί τη διεύθυνση και το μήκος του πίνακα που δημιουργήθηκε με την new, αυτές είναι οι πληροφορίες που αντιγράφονται στην table. Επομένως, η table δείχνει στον ίδιο πίνακα με αυτόν που δείχνει η array. Με άλλα λόγια αυτή η εκχώρηση δεν δημιουργεί αντίγραφο του πίνακα αλλά απλώς αντίγραφο της μεταβλητής διαχείρισης του πίνακα. Έτσι, όταν στη γραμμή 4 μεταβάλλουμε σε 9 την τιμή του στοιχείου στη θέση 1 του array και στη συνέχεια εμφανίζουμε το στοιχείο στη θέση 1 του table, βλέπουμε στην οθόνη μας να εμφανίζεται το 9. Είναι αποτέλεσμα του γεγονότος ότι και οι δύο μεταβλητές αναφέρονται στον ίδιο πίνακα στη μνήμη.
Επομένως, η προσέγγιση του κώδικα 6.5 δεν είναι κατάλληλη για τη δημιουργία ενός αντιγράφου πίνακα. Ωστόσο είναι εύκολο να κάνουμε έναν κώδικα που δημιουργεί αντίγραφο ενός πίνακα.

```
int[] array = {1, 3, 5};
int[] table = new int[array.length];
for (int i = 0; i < array.length; i++) {
    table[i] = array[i];
}
array[1] = 9;
System.out.println(table[1]);
```

Κώδικας 6.6 *Δημιουργία αντιγράφου μονοδιάστατου πίνακα*

Στον κώδικα 6.6 δημιουργούμε καταρχάς τον μονοδιάστατο πίνακα ακέραιων, array. Αμέσως μετά, δημιουργούμε επίσης έναν μονοδιάστατο πίνακα ακέραιων που ονομάζουμε table και έχει μήκος ίσο με το μήκος του array. Στη συνέχεια, με τη βοήθεια μιας for αντιγράφουμε ένα προς ένα τα στοιχεία του array στον table. Οι τελευταίες 2 γραμμές επιδεικνύουν ότι μεταβολές των στοιχείων του array δεν επηρεάζουν τα στοιχεία του table.

Η μέθοδος του κώδικα 6.6 για τη δημιουργία πινάκων μας δίνει μεν σωστό αντίγραφο, έχει όμως ένα μειονέκτημα. Για τη δημιουργία του αντιγράφου χρειάζεται να προσπελάσουμε, ένα προς ένα, όλα τα στοιχεία των δύο πινάκων. Αυτό στην περίπτωση που έχουμε πίνακες με μεγάλο μήκος στο πρόγραμμά μας μπορεί να θεωρηθεί χρονοβόρο. Με δεδομένο ότι τα στοιχεία ενός πίνακα είναι κατανεμημένα σε διαδοχικές θέσεις της μνήμης, θα ήταν πολύ ταχύτερο αν μπορούσαμε να αντιγράψουμε όλο το τμήμα της μνήμης στο οποίο είναι κατανεμημένος ο αρχικός πίνακας. Η Java μας παρέχει δύο συναρτήσεις με τις οποίες μπορούμε να παράγουμε αντίγραφα πινάκων ή τμημάτων πινάκων χωρίς να χρειάζεται να προσπελάσουμε ένα προς ένα τα επιμέρους στοιχεία. Πρόκειται για την arraycopy στατική συνάρτηση της κλάσης System και την copyOfRange στατική συνάρτηση της κλάσης Arrays.
Η γενική μορφή της arraycopy έχει ως εξής:

```
public static void arraycopy(Object source, int posAtSource,
Object destination, int posAtDest, int len)
```

Όπως φαίνεται από τη γενική μορφή, η συνάρτηση απαιτεί τις ακόλουθες παραμέτρους:

1. source είναι ο πίνακας του οποίου αντίγραφο θέλουμε να παράγουμε.
2. posAtSource είναι η θέση στον πίνακα source από την οποία επιθυμούμε να αρχίσει η αντιγραφή.
3. destination είναι ο πίνακας-αντίγραφο που θέλουμε να ενημερώσουμε.
4. posAtDest είναι η θέση στον πίνακα destination από όπου επιθυμούμε να αρχίσει η επικόλληση των στοιχείων του source.
5. len είναι το πλήθος των στοιχείων που επιθυμούμε να αντιγράψουμε.

Ο τύπος της πρώτης και της τρίτης παραμέτρου είναι Object. Ο τύπος αυτός παρουσιάζεται εκτεταμένα στην ενότητα 13.1.4. Για την ώρα, αρκεί να γνωρίζουμε ότι κατά την κλήση της arraycopy, μπορούμε να περάσουμε πίνακες στη θέση των παραμέτρων τύπου Object.

```
1   public static void arrayCopyDemo() { //Κώδικας 6.7
2       char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't',
'e', 'd'};
3       char[] copyTo = new char[7];
4        System.arraycopy(copyFrom, 2, copyTo, 0, copyTo.length);
5        System.out.println(new String(copyTo));
   }
```

Κώδικας 6.7 *Παράδειγμα χρήσης της arraycopy*

Στη γραμμή 2 του κώδικα 6.7 ορίζουμε τον πίνακα χαρακτήρων copyFrom. Όπως φανερώνει το όνομά του είναι ο πίνακας που θα αντιγραφεί και πιο συγκεκριμένα μέρος του οποίου θα αντιγράψουμε στον πίνακα copyTo που δηλώνεται και δημιουργείται στη γραμμή 3. Προσέξτε πως το μήκος του copyFrom είναι 13, ενώ του copyTo είναι 7. Η επιλογή αυτή οφείλεται στην πρόθεσή μας να αντιγράψουμε μόνο ένα τμήμα του copyFrom. Αν θέλουμε να αντιγράψουμε ολόκληρο τον πίνακα, τότε ο πίνακας προορισμού πρέπει να δημιουργηθεί σύμφωνα με το μήκος του πίνακα πηγής. Στη γραμμή 4 καλούμε την arraycopy για να πραγματοποιήσουμε την αντιγραφή. Στην κλήση αυτή περνάμε ως πρώτο όρισμα τον copyFrom και ως τρίτο τον copyTo. Το δεύτερο όρισμα σηματοδοτεί τη θέση στον copyFrom στην οποία αρχίζουν τα δεδομένα που θέλουμε να αντιγράψουμε. Εφόσον έχουμε περάσει την τιμή 2, τα δεδομένα θα αρχίσουν να αντιγράφονται από τη δεύτερη θέση του copyFrom, δηλαδή από τον χαρακτήρα ‘c’. Το τελευταίο όρισμα έχει ως αποτέλεσμα το πλήθος των χαρακτήρων που θα αντιγραφεί να είναι 7. Επομένως, στον copyTo θα αντιγραφούν οι χαρακτήρες 'c', 'a', 'f', 'f', 'e', 'i', 'n'. Τέλος, στη γραμμή 5 χρησιμοποιούμε μια βολική μέθοδο για να εκτυπώσουμε τον πίνακα χαρακτήρων copyTo. Αντί να εκτυπώσουμε έναν-έναν τους χαρακτήρες του, προφανώς με έναν κατάλληλο βρόχο, δημιουργούμε ένα String με όρισμα τον copyTo. Όπως ήδη γνωρίζουμε, η print γνωρίζει πώς ακριβώς να εμφανίσει δεδομένα τύπου String.

Σημαντικό είναι να προσέξουμε πως κατά τη χρήση της arraycopy πρέπει να δημιουργήσουμε έναν πίνακα με επαρκές μέγεθος για τα δεδομένα που θα αντιγραφούν. Στην περίπτωση που στη θέση του destination περάσουμε πίνακα μικρότερου μήκους από τα δεδομένα που θα αντιγραφούν θα προκληθεί σφάλμα χρόνου εκτέλεσης.

Ας δούμε τώρα την copyOfRange. Η διεπαφή (interface) της συνάρτησης για πίνακες ακέραιων έχει ως εξής:

```
public static int[] copyOfRange(int[] original, int from, int
to)
```

Το όρισμα int[] σημαίνει πως στη θέση εκείνη μπορούμε να περάσουμε οποιονδήποτε μονοδιάστατο πίνακα ακέραιων. Επίσης, το int[] πριν το όνομα της συνάρτησης σημαίνει πως η συνάρτηση θα επιστρέψει στο περιβάλλον κλήσης έναν μονοδιάστατο πίνακα ακέραιων.

Η συνάρτηση είναι υπερφορτωμένη (overloaded) για όλους τους θεμελιώδεις τύπους. Αυτό σημαίνει πως στη θέση του πίνακα ακέραιων μπορούμε να έχουμε πίνακα οποιουδήποτε θεμελιώδη τύπου. Περισσότερες λεπτομέρειες για την υπερφόρτωση συναρτήσεων στην ενότητα 7.2.

Επίσης, υποστηρίζεται μια γενίκευση (generic) της συνάρτησης η οποία μας επιτρέπει να αντιγράφουμε και πίνακες μη θεμελιωδών τύπων. Περισσότερα για τις γενικεύσεις στην ενότητα 17.

Συνοψίζοντας, σημειώνουμε πως στην copyOfRange μπορούμε να περνάμε πίνακες οποιουδήποτε τύπου. Πάντα θα μας επιστρέψει έναν πίνακα ίδιου τύπου με αυτόν που δώσαμε ως όρισμα κατά την κλήση της.

Τα ορίσματα from και to καθορίζουν από ποιο στοιχείο θα αρχίσει η αντιγραφή και σε ποιο στοιχείο θα καταλήξει. Ωστόσο, να δοθεί προσοχή στο γεγονός ότι το τελευταίο στοιχείο που θα αντιγραφεί είναι το στοιχείο στη θέση to-1 και όχι το στοιχείο στη θέση to.

```
    public static void copyOfRangeDemo() { //Κώδικας 6.8
        char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't',
'e', 'd'};
        char[] copyTo = Arrays.copyOfRange(copyFrom, 2, 9);
        System.out.println(new String(copyTo));
    }
```

Κώδικας 6.8 *Παράδειγμα χρήσης της copyOfRange*

Ο κώδικας 6.8 παρουσιάζει ένα παράδειγμα χρήσης της copyOfrange. Να υπογραμμίσουμε πως για να χρησιμοποιηθεί πρέπει να γίνει εισαγωγή της κλάσης Java.util.Arrays. Αυτό επιτυγχάνεται με την πρόταση import Java.util.Arrays; αμέσως μετά τη δήλωση του πακέτου. Περισσότερα για τη διαδικασία import στην ενότητα 7.8.

Αρχικά ορίζουμε τον πίνακα χαρακτήρων copyFrom. Στη συνέχεια καλούμε την copyOfRange με ορίσματα τον copyFrom, το 2 και το 9. Η copyOfRange θα δημιουργήσει και θα επιστρέψει έναν πίνακα χαρακτήρων με δεδομένα από τη θέση 2 του copyFrom, δηλαδή από τον χαρακτήρα 'c' έως και τη θέση 8, δηλαδή τον χαρακτήρα 'n'. Επομένως, η copyOfRangeDemo θα εμφανίσει τη σειρά "caffein".

Προσέξτε ότι κατά τη χρήση της copyOfRange δεν χρειάστηκε να δημιουργήσουμε πίνακα στον οποίο θα αντιγράφονταν τα δεδομένα. Τον πίνακα αυτόν δημιούργησε η copyOfRange και τον επέστρεψε στο πρόγραμμά μας. Επομένως, κατά τη χρήση της copyOfRange δεν κινδυνεύουμε να κάνουμε λάθος με το μέγεθος του πίνακα-αντιγράφου όπως στην arraycopy.

6.2 Η κλάση Arrays

Εκτός από την copyOfRange, η κλάση Arrays περιλαμβάνει μια σειρά από στατικές συναρτήσεις που υποστηρίζουν ποικιλία επεξεργασιών των πινάκων. Παρόμοια με την copyOfRange, οι συναρτήσεις που συζητάμε εδώ είναι οργανωμένες κατά τέτοιο τρόπο ώστε στις περισσότερες περιπτώσεις να επιτρέπουν την επεξεργασία πίνακα οποιουδήποτε τύπου. Σε αυτήν την ενότητα θα παρουσιάσουμε μέρος των συναρτήσεων της Arrays. Η διεπαφή των συναρτήσεων παρουσιάζεται για πίνακες ακέραιων αλλά αντίστοιχα ισχύουν και για τους υπόλοιπους τύπους. Όπου υπάρχουν διαφορές επισημαίνονται σαφώς.

6.2.1 Ταξινόμηση

Όλοι οι θεμελιώδεις τύποι πλην του τύπου boolean έχουν στην Java μια φυσική σειρά (natural order), δηλαδή για 2 τιμές ενός τύπου είναι προκαθορισμένο ποια είναι μεγαλύτερη και ποια μικρότερη. Οι τιμές boolean θεωρείται πως δεν διαθέτουν φυσική σειρά, δηλαδή δεν θεωρείται εξ ορισμού πως το false είναι μικρότερο από το true ή το αντίθετο.

Η Java υποστηρίζει ταξινόμηση πινάκων όλων των θεμελιωδών, πλην του boolean, τύπων καθώς και των μη θεμελιωδών τύπων. Ωστόσο, για πολλούς μη θεμελιώδεις τύπους δεν υπάρχει φυσική σειρά. Αντίθετα, η σειρά των μη θεμελιωδών τύπων πρέπει να καθοριστεί με ευθύνη του προγραμματιστή. Το θέμα αυτό θα συζητήσουμε στην ενότητα 14.1.2. Η συζήτηση εδώ θα περιοριστεί στην ταξινόμηση πινάκων θεμελιωδών τύπων. Η διεπαφή της συνάρτησης ταξινόμησης για πίνακες ακέραιων είναι:

```
public static void sort(int[] a)
```

Επομένως, αν επιθυμούμε να ταξινομήσουμε έναν πίνακα ακέραιων t, αρκεί η κλήση:

```
Arrays.sort(t);
```

Ίδια είναι η κλήση και για οποιονδήποτε άλλο θεμελιώδη τύπο, πλην του τύπου boolean.

6.2.2 Δυαδική αναζήτηση (Binary Search)

Κατά την αναζήτηση, ελέγχουμε έναν πίνακα για να διαπιστώσουμε αν περιλαμβάνει ή όχι μια τιμή. Αν η τιμή βρεθεί, επιστρέφεται η θέση της στον πίνακα. Αν η τιμή δεν βρεθεί, επιστρέφεται -1. Ειδικότερα, η δυαδική αναζήτηση εφαρμόζεται μόνο σε ταξινομημένους πίνακες. Η εφαρμογή της σε μη ταξινομημένους έχει απροσδιόριστα αποτελέσματα.
Η διεπαφή της συνάρτησης δυαδικής αναζήτησης για πίνακες ακέραιων έχει ως εξής:

```
public static int binarySearch(int[] a, int key)
```

Η binarySearch λαμβάνει ως παραμέτρους τον πίνακα στον οποίο επιθυμούμε να πραγματοποιήσουμε αναζήτηση και την τιμή που αναζητούμε. Επιστρέφει τη θέση στον πίνακα του στοιχείου που αναζητούμε ή -1 σε περίπτωση που το στοιχείο δεν υπάρχει στον πίνακα. Επομένως, αν έχουμε έναν μονοδιάστατο ταξινομημένο πίνακα ακέραιων, έστω t, και ψάχνουμε για το στοιχείο schElement, η κλήση θα πρέπει να είναι του τύπου:

```
int idx=Arrays.binarySearch(t, schElement);
```

6.2.3 Έλεγχος Ισότητας

Δύο πίνακες θεωρούνται ίσοι μεταξύ τους αν περιέχουν τα ίδια στοιχεία και με την ίδια σειρά.
Η διεπαφή της συνάρτησης ισότητας για πίνακες ακέραιων έχει ως εξής:

```
public static boolean equals(int[] a, int[] a2)
```

Επομένως, αν έχουμε τους πίνακες t1 και t2 και θέλουμε να τους συγκρίνουμε για ισότητα αρκεί η κλήση:

```
boolean tEquals=Arrays.equals(t1, t2);
```

6.2.4 Αρχικοποίηση

Η αρχικοποίηση πινάκων με την ίδια τιμή για κάθε στοιχείο του πίνακα γίνεται με τις συναρτήσεις fill. Η διεπαφή της fill για πίνακες ακέραιων έχει ως εξής:

```
public static void fill(int[] a, int val)
```

Επομένως, για να αρχικοποιήσουμε όλα τα στοιχεία ενός πίνακα ακέραιων, t, στην τιμή 1, αρκεί η κλήση

```
Arrays.fill(t,1);
```

6.2.5 Αναπαράσταση πίνακα ως String

Αν χρειαζόμαστε αναπαράσταση ενός πίνακα ως αλφαριθμητική σειρά μπορούμε να χρησιμοποιήσουμε την ομάδα συναρτήσεων toString. Η διεπαφή της συνάρτησης για πίνακες ακέραιων έχει ως εξής:

```
public static String toString(int[] a)
```

Για παράδειγμα, αν θέλουμε να τυπώσουμε τον πίνακα ακέραιων t, αρκεί ο ακόλουθος κώδικας:

```
System.out.println(Arrays.toString(t));
```

6.3 Πολυδιάστατοι πίνακες

Όπως αναφέραμε στην αρχή αυτού του κεφαλαίου, η Java υποστηρίζει και πολυδιάστατους πίνακες. Θα παρουσιάσουμε εδώ κυρίως τους δισδιάστατους πίνακες ενώ θα δώσουμε και παραδείγματα χρήσης πινάκων με περισσότερες διαστάσεις. Πάντως, ό,τι ισχύει για τους δισδιάστατους πίνακες, ισχύει αναλογικά και για πίνακες περισσότερων διαστάσεων.
Ένας πίνακας δύο διαστάσεων ενός τύπου στην Java είναι ένας πίνακας κάθε στοιχείο του οποίου είναι ένας μονοδιάστατος πίνακας του ίδιου τύπου.

```
static void twoDimArray() { //Κώδικας 6.9
    int[][] store = {
        {1, 2, 3, 4},
        {5, 6, 7},
        {8, 9, 10, 11, 12}
    };
    System.out.println(Arrays.toString(store[0]));
    System.out.println(Arrays.toString(store[1]));
    System.out.println(Arrays.toString(store[2]));
}
```

Κώδικας 6.9 *Παράδειγμα δήλωσης και αρχικοποίησης πίνακα δύο διαστάσεων με αρχικοποιητή*

Στο παράδειγμα του κώδικα 6.9 δηλώνουμε και αρχικοποιούμε τον δισδιάστατο πίνακα ακέραιων store. Ο πίνακας έχει δηλωθεί ως δισδιάστατος, καθώς τη δήλωση του τύπου int ακολουθούν δύο σύνολα από τετραγωνικές παρενθέσεις. Αντίστοιχα, μπορούμε να δηλώσουμε τρισδιάστατους πίνακες χρησιμοποιώντας τρία σύνολα τετραγωνικών παρενθέσεων, τετραδιάστατους με τέσσερα σύνολα, κοκ, χωρίς άνω όριο διαστάσεων. Στο συγκεκριμένο παράδειγμα, ο πίνακας store αρχικοποιήθηκε με τη χρήση αρχικοποιητή.
Τα στοιχεία του store είναι τρεις μονοδιάστατοι πίνακες ακέραιων. Επομένως, το μήκος του store είναι τρία, store.length==3. Τα δε στοιχεία του είναι τα εξής: το store[0] είναι ο πίνακας {1, 2, 3, 4}, το store[1] είναι ο πίνακας {5, 6, 7} και το store[2] είναι ο πίνακας {8, 9, 10, 11, 12}. Προσέξτε πως κάθε ένας από τους πίνακες-στοιχεία του store έχει διαφορετικό μέγεθος.

```
static void createTwoDimWithNew() { //Κώδικας 6.10
    int[][] store = new int[3][];
    store[0] = new int[4];
    store[1] = new int[3];
    store[2] = new int[5];
    int data = 0;
    for (int i = 0; i < store.length; i++) {
        for (int j = 0; j < store[i].length; j++) {
            store[i][j] = 1 + data++;
        }
    }
    System.out.println(Arrays.toString(store[0]));
    System.out.println(Arrays.toString(store[1]));
    System.out.println(Arrays.toString(store[2]));
}
```

Κώδικας 6.10 *Παράδειγμα δήλωσης και αρχικοποίησης πίνακα δύο διαστάσεων με χρήση της new*

Στον κώδικα 6.10 δημιουργούμε έναν δισδιάστατο πίνακα με τη βοήθεια της new. Στην περίπτωση αυτή, η μία διάσταση, αυτή που καθορίζει τα στοιχεία του store, υποχρεωτικά δηλώνεται μέσω μιας σταθεράς. Η

δεύτερη μπορεί να δηλωθεί ή να μην δηλωθεί. Αν δηλωθεί όμως, τότε υποχρεωτικά όλα τα στοιχεία-πίνακες του store θα έχουν τη διάσταση που δηλώθηκε κατά την εκτέλεση της new. Στο παράδειγμά μας, δεν έχουμε δηλώσει τη δεύτερη διάσταση. Έτσι μπορούμε να δημιουργήσουμε στοιχεία-πίνακες του store ποικίλων μεγεθών. Στην περίπτωση αυτή, τα στοιχεία του store πρέπει να δημιουργηθούν ένα προς ένα με χρήση της new. Για τα στοιχεία αυτά δεν είναι δυνατή η χρήση αρχικοποιητή, καθώς ο αρχικοποιητής είναι αποδεκτός μόνο στη γραμμή δήλωσης ενός πίνακα.

Στη συνέχεια, στον κώδικα 6.10, προσπελαύνουμε ένα προς ένα τα τελικά στοιχεία του store με μια εμφωλευμένη for προκειμένου να τους καταχωρήσουμε τιμές. Προσέξτε πως η εσωτερική for πηγαίνει μέχρι store[i].length. Συχνά παρατηρείται σε κώδικες αρχαρίων να χρησιμοποιείται το μήκος του δισδιάστατου πίνακα ως άνω όριο της εσωτερικής for. Αυτό είναι προφανώς λανθασμένη πρακτική, καθώς είναι δυνατό ο δισδιάστατος πίνακας να έχει διαφορετικό μήκος από τα στοιχεία-πίνακες που περιέχει.

Αν όμως όλα τα στοιχεία-πίνακες ενός δισδιάστατου πίνακα έχουν την ίδια διάσταση, τότε αν την δηλώσουμε, τα στοιχεία-πίνακες δημιουργούνται αυτόματα χωρίς τη χρήση της new για καθένα από αυτά.

```
static void createArrayWithCommonLength() { //Κώδικας 6.11
    int[][] s = new int[3][3];
    int data = 0;
    for (int i = 0; i < s.length; i++) {
        for (int j = 0; j < s[i].length; j++) {
            s[i][j] = 1 + data++;
        }
    }
    System.out.println(Arrays.toString(s[0]));
    System.out.println(Arrays.toString(s[1]));
    System.out.println(Arrays.toString(s[2]));
}
```

Κώδικας 6.11 *Δημιουργία δισδιάστατου πίνακα με σταθερό μήκος στοιχείων*

Στον κώδικα 6.11 τα στοιχεία του δισδιάστατου πίνακα s έχουν όλα το ίδιο μήκος. Σε αυτήν την περίπτωση κατά τη δημιουργία του s δηλώνουμε και τη δεύτερη διάσταση, οπότε ο s και τα στοιχεία του δημιουργούνται χωρίς να απαιτείται η αρχικοποιητή χρήση της new για κάθε στοιχείο ξεχωριστά.

Θα θεωρήσει κανείς πως σε αυτήν την περίπτωση δεν είναι αναγκαίο η εσωτερική for να αναφέρεται στο μήκος του τρέχοντος στοιχείου. Μια και το μήκος κάθε στοιχείου είναι ίδιο με το μήκος του s μπορεί να χρησιμοποιηθεί το μήκος του s και στην εσωτερική for. Αυτή η προσέγγιση είναι λανθασμένη. Η κατάσταση του πίνακα ανεξάρτητα από το πώς δημιουργήθηκε δεν είναι σταθερή. Είναι πιθανό να έχουν μεσολαβήσει μεταβολές από τη δημιουργία του μέχρι την επεξεργασία του. Για παράδειγμα, μετά τη δημιουργία του s είναι δυνατό να τρέξει κώδικας της μορφής s[1]=new int[7];. Ο κώδικας αυτός μεταβάλλει τον πίνακα στη θέση 1 του s. Για αυτόν τον λόγο, σε όλες τις περιπτώσεις, θα πρέπει να θυμόμαστε πως το μήκος ενός πίνακα, αξιόπιστα, το γνωρίζει μόνο ο ίδιος ο πίνακας.

Σε ποιες περιπτώσεις μπορούμε να αξιοποιήσουμε έναν πίνακα τριών διαστάσεων; Εξαρτάται από τα δεδομένα του προβλήματος που έχουμε να κωδικοποιήσουμε. Έστω, ότι έχουμε δεδομένα σχετικά με τις ώρες απασχόλησης ενός εργαζόμενου. Ας υποθέσουμε πως τα δεδομένα μας είναι δομημένα σε τέσσερις χρονικές περιόδους κάθε μία από τις οποίες αποτελείται από τρεις μήνες, κάθε ένας από τους οποίους περιλαμβάνει είκοσι δύο εργάσιμες ημέρες. Για κάθε μία από αυτές τις ημέρες, ο εργαζόμενος έχει εργαστεί έναν ορισμένο αριθμό ωρών. Τέτοιου τύπου δεδομένα είναι πολύ βολικό να φορτωθούν σε έναν τρισδιάστατο πίνακα. Για παράδειγμα, έστω ο πίνακας

```
int[][][] timesheet=new int[4][3][22];
```

Η πρώτη διάσταση αφορά τις περιόδους, η δεύτερη τον μήνα μιας περιόδου και η τρίτη, τις εργάσιμες ημέρες ενός μήνα. Επομένως, αν θέλουμε να προσπελάσουμε πόσες ώρες εργάστηκε ο εργαζόμενος την έβδομη ημέρα του δεύτερου μήνα της πρώτης περιόδου αρκεί να προσπελάσουμε το στοιχείο timesheet[0][1][6].

6.4 Βασικές επεξεργασίες

Συχνά απαιτείται να κάνουμε κάποιες βασικές εργασίες με πίνακες όπως να υπολογίσουμε το άθροισμα των στοιχείων ενός πίνακα αριθμών, να εντοπίσουμε το μεγαλύτερο ή το μικρότερο στοιχείο ενός πίνακα, να πραγματοποιήσουμε σειριακή αναζήτηση ψάχνοντας για μια συγκεκριμένη τιμή σε έναν πίνακα, να ανακατέψουμε τα στοιχεία ενός πίνακα, να γεμίσουμε έναν πίνακα με τυχαίες τιμές. Σε αυτήν την ενότητα, θα δούμε τους κώδικες που μπορούν να αξιοποιηθούν για την υλοποίηση των αναφερόμενων επεξεργασιών.

6.4.1 Σειριακή αναζήτηση

Η σειριακή αναζήτηση σε αντίθεση με τη δυαδική μπορεί να εφαρμοστεί σε αταξινόμητους πίνακες. Όπως φανερώνει το όνομά της, ελέγχει ένα προς ένα τα στοιχεία του πίνακα συγκρίνοντάς τα με το στοιχείο που ψάχνουμε.
Ο κώδικας της σειριακής αναζήτησης είναι σχετικά απλός:

```
    public static void sequentialSearch() { //Κώδικας 6.12
        final int[] store = {4, 7, 9, 13};
        int schElement = 7;
        int idx = -1;
        for (int i = 0; i < store.length; i++) {
            if (store[i] == schElement) {
                idx = i;
                break;
            }
        }
        if (idx >= 0) {
            System.out.println("Ο ακέραιος " + schElement + " βρέθηκε στην θέση "
+ idx);
        } else {
            System.out.println("Ο ακέραιος " + schElement + " δεν βρέθηκε");
        }
    }
```

Κώδικας 6.12 *Σειριακή αναζήτηση σε μονοδιάστατο πίνακα ακέραιων*

Στον κώδικα 6.12 ορίζουμε καταρχάς τον πίνακα ακέραιων store και το στοιχείο για το οποίο θα ψάξουμε, το schElement. Επίσης, ορίζουμε μια ακέραιη μεταβλητή idx που αρχικοποιούμε στο -1. Αν κατά την αναζήτηση στον πίνακα βρεθεί το schElement, τότε στην idx θα εκχωρήσουμε τη θέση του στον πίνακα, οπότε η idx θα αποκτήσει τιμή μεγαλύτερη ή ίση με το 0. Έτσι θα γνωρίζουμε πως το στοιχείο βρέθηκε αλλά και πού ακριβώς είναι τοποθετημένο. Αν με το τέλος της αναζήτησης το στοιχείο δεν έχει βρεθεί, η idx θα παραμείνει μικρότερη από το μηδέν σηματοδοτώντας την αποτυχία εύρεσης του στοιχείου. Όπως έχουμε ήδη δει, η σύμβαση σύμφωνα με την οποία επιστρέφεται -1 για ένα στοιχείο που δεν βρέθηκε σε ένα πίνακα χρησιμοποιείται και από τη συνάρτηση binarySearch της κλάσης Arrays που παρουσιάστηκε στην 6.2.2.

Αφού λοιπόν ορίσουμε την idx μπαίνουμε σε έναν βρόχο for με τη βοήθεια του οποίου ελέγχουμε ένα προς ένα τα στοιχεία του store συγκρίνοντάς τα με το schElement. Αν για κάποιο από αυτά επιβεβαιωθεί ισότητα, τότε το schElement βρέθηκε, οπότε ενημερώνεται η idx και διακόπτεται η επαναληπτική διαδικασία.

Σε μια τέτοια συνάρτηση θα πρέπει να προσέξουμε ιδιαίτερα τον τύπο του πίνακα. Το πρότυπο του κώδικα 6.12 είναι κατάλληλο για πίνακες θεμελιωδών τύπων, όχι όμως και για πίνακες μη θεμελιωδών τύπων όπως ο τύπος String. Στην περίπτωση αυτή ο τελεστής ισότητας == θα πρέπει να αντικατασταθεί με τη συνάρτηση equals του τύπου του πίνακα όπως δείχνει ο κώδικας 6.13:

```
    public static void sequentialSearchReference() { //Κώδικας 6.13
        final String[] store = {"John", "Mary", "Jim", "Kelly"};
        String schElement = new String("Jim");
        int idx = -1;
        for (int i = 0; i < store.length; i++) {
            if (store[i].equals(schElement)) {
                idx = i;
                break;
            }
```

```
        }
        if (idx >= 0) {
            System.out.println("Το όνομα " + schElement + " βρέθηκε στην θέση " +
idx);
        } else {
            System.out.println("Το όνομα " + schElement + " δεν βρέθηκε");
        }
    }
```

Κώδικας 6.13 *Σειριακή αναζήτηση σε πίνακα αλφαριθμητικών σειρών*

6.4.2 Μέγιστο ή ελάχιστο στοιχείο

Ένα άλλο πρόβλημα είναι η αναζήτηση του μέγιστου στοιχείου ενός πίνακα. Πρόκειται για απλή επεξεργασία. Παραθέτουμε τον κώδικα υλοποίησης και στη συνέχεια τον σχολιάζουμε:

```
    public static void max() { //Κώδικας 6.14
        int[] t = {2, 6, 1, 9, 15};
        int max = t[0];
        int idx = 0;
        for (int i = 1; i < t.length; i++) {
            if (t[i] > max) {
                max = t[i];
                idx = i;
            }
        }
        System.out.println("Το μεγαλύτερο στοιχείο είναι το " + max + " και
βρίσκεται στην θέση " + idx);
```

Κώδικας 6.14 *Εύρεση του μεγαλύτερου στοιχείου μονοδιάστατου πίνακα*

Στον κώδικα 6.14 καταρχάς ορίζουμε τον πίνακα t. Στη συνέχεια, δύο βοηθητικές μεταβλητές, την max και την idx. Η max αρχικοποιείται στην τιμή του στοιχείου στη θέση μηδέν του πίνακα. Επίσης, η idx αρχικοποιείται ώστε να δείχνει στη θέση μηδέν. Ακολούθως, προσπελαύνουμε τα στοιχεία του πίνακα ένα προς ένα και όποτε βρίσκουμε τιμή μεγαλύτερη από την τρέχουσα τιμή του max ενημερώνουμε την max και την idx. Στο τέλος της διαδικασίας, η max έχει τη μεγαλύτερη τιμή του πίνακα και η idx τη θέση της.

Προσέξτε πως ο βρόχος for ξεκινάει από 1 και όχι από 0. Πράγματι, στο συγκεκριμένο πρόβλημα δεν υπάρχει λόγος να ξεκινήσει η σύγκριση από το 0. Αντίθετα, αν ξεκινούσε από το 0, κατά το πρώτο επαναληπτικό βήμα θα συγκρινόταν η max με το στοιχείο t[0]. Η max όμως σε αυτό το βήμα έχει τιμή ίση με το t[0], οπότε είναι περιττή η σύγκριση.

Στην περίπτωση που ο πίνακας έχει δύο στοιχεία ίσα μεταξύ τους και μεγαλύτερα από τα υπόλοιπα στοιχεία του πίνακα, ο κώδικας 6.14 εντοπίζει το στοιχείο στη μικρότερη θέση. Αν επιθυμούμε να βρούμε το στοιχείο στη μεγαλύτερη θέση αρκεί να τροποποιήσουμε τη συνθήκη της if σε t[i]>=max.

Αντίστοιχα, η εύρεση του μικρότερου στοιχείου, βρίσκεται αν αντικαταστήσουμε τη συνθήκη ελέγχου της if με την έκφραση t[i]<min ή t[i]<=min.

6.4.3 Άθροισμα

Ένα άλλο τυπικό πρόβλημα είναι ο υπολογισμός του αθροίσματος των στοιχείων ενός πίνακα. Πρόκειται επίσης για μια σειριακή προσπέλαση των στοιχείων του πίνακα και προσαύξηση ενός αθροιστή κατά την τιμή κάθε στοιχείου. Ο κώδικας 6.15 παρουσιάζει τη λύση.

```
    public static void sum() { //Κώδικας 6.15
        int[] t = {2, 6, 1, 9, 15};
        int sum = 0;
        for (int i = 0; i < t.length; i++) {
            sum += t[i];
        }
        System.out.println("Το άθροισμα είναι " + sum);
```

```
}
```

Κώδικας 6.15 *Άθροισμα στοιχείων μονοδιάστατου πίνακα*

6.4.4 Ανακατανομή

Σε πολλές εφαρμογές απαιτείται τυχαία ανακατανομή των στοιχείων ενός πίνακα. Για κάθε στοιχείο του πίνακα, υπολογίζουμε μια τυχαία θέση μέσα στα όρια του πίνακα και αντιμεταθέτουμε το στοιχείο στην τρέχουσα θέση με το στοιχείο στην τυχαία θέση.

```
public static void shuffle() { //Κώδικας 6.16
    int[] t = {2, 6, 1, 9, 15};
    Arrays.sort(t);
    System.out.println(Arrays.toString(t));
    Random r = new Random();

    for (int i = 0; i < t.length; i++) {
        int rIdx = r.nextInt(t.length);
        int tmp = t[i];
        t[i] = t[rIdx];
        t[rIdx] = tmp;
    }
    System.out.println(Arrays.toString(t));
}
```

Κώδικας 6.16 *Ανακατανομή στοιχείων πίνακα*

Στον κώδικα 6.16 δημιουργούμε το πίνακα t, τον ταξινομούμε και τον τυπώνουμε έτσι ώστε να είναι εμφανής η διαφορά από τον ανακατεμένο πίνακα. Στη συνέχεια υλοποιείται ο αλγόριθμος ανακατανομής που περιγράφηκε αμέσως προηγουμένως.

6.4.5 Αρχικοποίηση με τυχαίες τιμές

Πολλές φορές χρειαζόμαστε να γεμίσουμε έναν πίνακα με τυχαίες τιμές είτε για λόγους ελέγχου είτε επειδή η εφαρμογή μας το απαιτεί. Στον κώδικα 6.17 ο πίνακας γεμίζει με τυχαίες τιμές από το 0 έως 99.

```
public static void fillRandom() { //Κώδικας 6.17
    int[] t = new int[20];
    Random r = new Random();
    for (int i = 0; i < t.length; i++) {
        t[i] = r.nextInt(100);
    }
    System.out.println(Arrays.toString(t));
}
```

Κώδικας 6.17 *Αρχικοποίηση πίνακα με τυχαίες τιμές*

6.4.6 Τελικοί πίνακες

Η δεσμευμένη λέξη final μπορεί να χρησιμοποιηθεί και με μεταβλητές που αναφέρονται σε πίνακες. Η σημασία της είναι ακριβώς ίδια με τη σημασία που έχει όταν χρησιμοποιείται με μεταβλητές τιμής. Ωστόσο, επειδή οι μεταβλητές αναφοράς έχουν διαφορετική συμπεριφορά, συχνά παρατηρείται σύγχυση στον αρχάριο προγραμματιστή σε σχέση με τις επιπτώσεις του χαρακτηρισμού ως final ενός πίνακα.

Όπως αναφέρουμε στην ενότητα 3.1.2, ο χαρακτηρισμός ως final μιας μεταβλητής έχει ως συνέπεια να επιτρέπεται εκχώρηση στη μεταβλητή μόνο μία φορά. Το ίδιο ακριβώς συμβαίνει και με τις μεταβλητές πινάκων. Επομένως, ο κώδικας που ακολουθεί παράγει λάθος χρόνου μεταγλώττισης.

```
final int[] t={1,3,5,7};
t=new int[9];
```

Ωστόσο, η μεταβολή των στοιχείων του πίνακα είναι διαφορετική υπόθεση. Αυτή μπορεί να γίνει χωρίς να μεταβληθεί η αναφορά στον πίνακα.

Επομένως, ο ακόλουθος κώδικας είναι απολύτως σωστός:

```
final int[] t={1,3,5,7};
t[0]=10;
```

6.5 Λυμένες Ασκήσεις

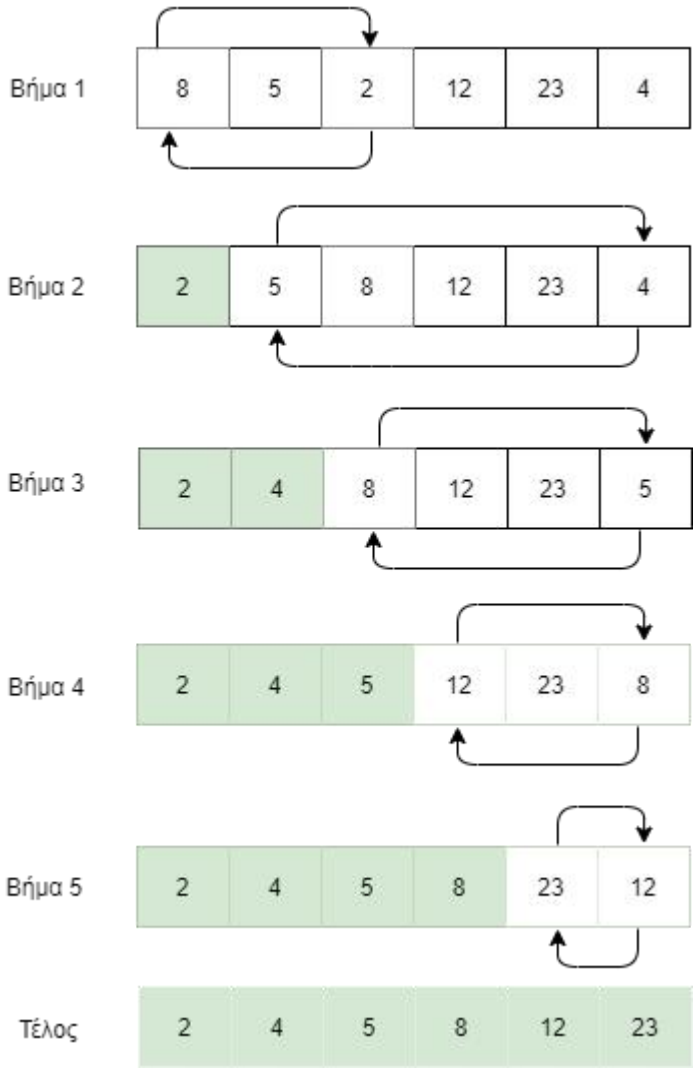
Παρουσιάζουμε εδώ μια σειρά από βασικές επεξεργασίες πινάκων.

6.5.1 Ταξινόμηση

Να αναπτύξετε συνάρτηση στην οποία να ορίσετε έναν πίνακα ακέραιων και στη συνέχεια να τον ταξινομήσετε κατ' αύξουσα σειρά χωρίς να χρησιμοποιήσετε συνάρτηση sort της Arrays.

Λύση

Υπάρχουν πολλοί αλγόριθμοι ταξινόμησης δεδομένων. Για τη λύση αυτής της άσκησης θα χρησιμοποιήσουμε ίσως τον απλούστερο όλων, γνωστό ως selection sort [1]. Το βασικό βήμα σε αυτήν την ταξινόμηση είναι η ανταλλαγή θέσης μεταξύ μικρότερου στοιχείου και του στοιχείου που είναι στην αρχή του αταξινόμητου μέρους του πίνακα. Το βήμα επαναλαμβάνεται από το πρώτο μέχρι το προτελευταίο στοιχείο του πίνακα. Με το τέλος της επανάληψης ο πίνακας είναι ταξινομημένος. Εξηγούμε λεπτομερέστερα με τη βοήθεια του σχήματος 6.2.



Σχήμα 6.2 *Αναπαράσταση του selection sort*

Βήμα 1: Ολόκληρος ο πίνακας είναι αταξινόμητος. Το πρώτο στοιχείο του αταξινόμητου πίνακα βρίσκεται στη θέση 0 και έχει τιμή 8. Το μικρότερο στοιχείο του πίνακα βρίσκεται στη θέση 2 και έχει τιμή 2. Τα στοιχεία θα ανταλλάξουν θέσεις.

Βήμα 2: Μετά την ανταλλαγή θέσεων του βήματος 1, το μικρότερο στοιχείο βρίσκεται στη θέση 0. Το τμήμα του πίνακα που περιλαμβάνει τη θέση 0 θεωρείται ταξινομημένο. Το μη ταξινομημένο τμήμα του πίνακα εκτείνεται από τη θέση 1 έως το τέλος του πίνακα. Επαναλαμβάνεται η λογική του βήματος 1 για το αταξινόμητο τμήμα του πίνακα. Το μικρότερο στοιχείο με τιμή 4 θα ανταλλάξει θέση με το στοιχείο με τιμή 5.

Βήμα 3: Με την ίδια λογική το 5 θα ανταλλάξει θέση με το 8.

Βήμα 4: Το 8 θα ανταλλάξει θέση με το 12

Βήμα 5: Το 12 θα ανταλλάξει θέση με το 23

Τέλος: Ο πίνακας είναι ταξινομημένος

Στην συνέχεια δίνουμε την υλοποίηση της selectionSort:

```
1   static void selectionSort(String[] args) { //Κώδικας 6.18
2       int[] tbl = {2, 1, 6, 3, 4, 9, 8, 5, 7, 4};
3       for (int i = 0; i < tbl.length - 1; i++) {
4           int min = tbl[i];
5           int idx = i;
```

```
 6            for (int j = i + 1; j < tbl.length; j++) {
 7                if (tbl[j] < min) {
 8                    min = tbl[j];
 9                    idx = j;
10                }
11            }
12
13            if (idx != i) {
14               int tmp = tbl[i];
15               tbl[i] = tbl[idx];
16               tbl[idx] = tmp;
17            }
18        }
19        System.out.println(Arrays.toString(tbl));
20    }
```

Κώδικας 6.18 *Υλοποίηση του selection sort*

Στη γραμμή 2 ορίζουμε έναν πίνακα ακέραιων. Στη γραμμή 3 ξεκινάμε τη βασική επαναληπτική διαδικασία. Στις γραμμές 4 έως 11 αναζητούμε το μικρότερο στοιχείο του πίνακα από τη θέση i και μετά. Αν το μικρότερο στοιχείο βρίσκεται σε θέση διαφορετική από το i, αντιμεταθέτουμε τα δύο στοιχεία (γραμμές 11 έως 15). Ο εξωτερικός βρόχος (γραμμή 3) ελέγχει μέχρι tbl.length-2. Αυτό είναι λογικό, καθώς το tbl.length-1 είναι η τελευταία θέση του πίνακα και δεν έχει νόημα να ελέγξουμε αν μετά από αυτήν υπάρχει στοιχείο με μικρότερη τιμή. Στη γραμμή 19 εμφανίζουμε τον ταξινομημένο πίνακα.

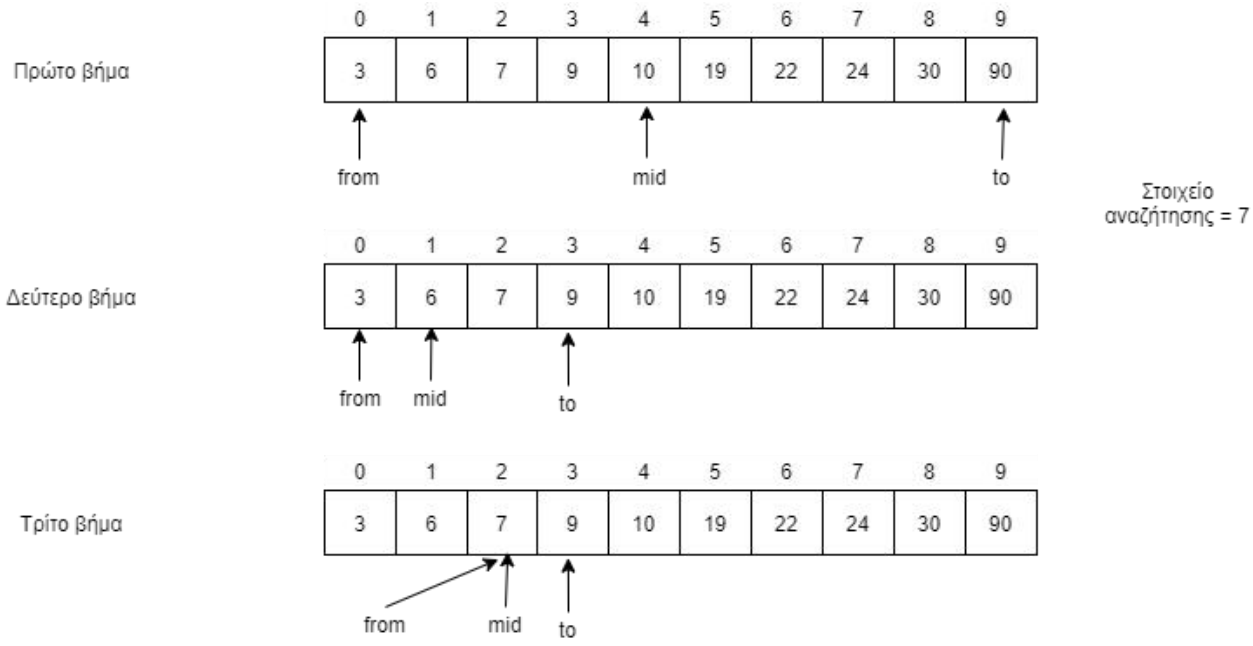
6.5.2 Δυαδική αναζήτηση

Να αναπτύξετε συνάρτηση Java στην οποία να ορίσετε έναν μονοδιάστατο πίνακα ακέραιων, έναν ακέραιο και στη συνέχεια να πραγματοποιήσετε δυαδική αναζήτηση στον πίνακα και να βρείτε τον ακέραιο χωρίς να χρησιμοποιήσετε τις συναρτήσεις δυαδικής αναζήτησης της Arrays.

Λύση

Η βασική ιδέα της δυαδικής αναζήτησης (binary search) [2] είναι να εξετάζουμε κάθε φορά ένα τμήμα του πίνακα για το οποίο ελέγχουμε αν η τιμή που αναζητούμε είναι μεγαλύτερη, μικρότερη ή ίση από την τιμή στο μέσο του τμήματος. Αν είναι ίση, το στοιχείο αναζήτησης βρέθηκε. Αν είναι μικρότερη, μας μένει να ψάξουμε μόνο στο πρώτο μισό μέρος του τμήματος του πίνακα. Αν είναι μεγαλύτερη, μας μένει να ψάξουμε μόνο στο δεύτερο μισό μέρος του τμήματος του πίνακα. Με αυτόν τον τρόπο, η δυαδική αναζήτηση υποδιπλασιάζει σε κάθε επαναληπτικό βήμα τα στοιχεία του πίνακα που μένουν προς έλεγχο. Κατά το πρώτο βήμα, το τμήμα του πίνακα που έχουμε να ψάξουμε συμπίπτει με ολόκληρο τον πίνακα.

Για να υλοποιήσουμε τη δυαδική αναζήτηση χρειαζόμαστε πέραν του πίνακα και του στοιχείου αναζήτησης και δύο δείκτες, ας τους ονομάσουμε from και to, που καθορίζουν τα όρια του τμήματος που ελέγχουμε σε κάθε επαναληπτικό βήμα. Στο σχήμα 6.3 παρουσιάζεται παράδειγμα δυαδικής αναζήτησης:



Σχήμα 6.3 *Παράδειγμα δυαδικής αναζήτησης*

Στην αρχή της αναζήτησης, ο δείκτης from δείχνει στην αρχή του πίνακα και ο δείκτης to στο τέλος. Συμπεριλαμβάνεται έτσι όλος ο πίνακας στην αναζήτηση. Με βάση τις τιμές των from και to υπολογίζεται το μέσο, mid, του πίνακα. Στη συνέχεια, εφόσον το στοιχείο αναζήτησης είναι μεγαλύτερο από την τιμή στο μέσο του πίνακα και ο πίνακας είναι ταξινομημένος κατ' αύξουσα σειρά, εξάγεται το συμπέρασμα πως το στοιχείο αναζήτησης, αν υπάρχει, θα πρέπει να αναζητηθεί στο δεύτερο τμήμα του πίνακα, δηλαδή στο τμήμα που αρχίζει από mid+1 και το τέλος του συμπίπτει με το τέλος του πίνακα. Αντίθετα, αν όπως στο παράδειγμά μας, το στοιχείο αναζήτησης είναι μικρότερο από την τιμή στο μέσο του πίνακα, τότε η αναζήτηση θα πρέπει να συνεχίσει με το πρώτο μισό τμήμα του πίνακα, δηλαδή να εφαρμοστεί στο τμήμα from έως mid-1.

Έτσι στο δεύτερο βήμα, ο δείκτης to γίνεται ίσος με mid-1 και άρα το νέο μέσο τοποθετείται στη θέση 1 του πίνακα και δείχνει μια θέση μετά το μέσο του πίνακα. Στη συνέχεια, το στοιχείο αναζήτησης είναι μεγαλύτερο από την τιμή στη θέση 1, οπότε ο δείκτης from γίνεται mid+1. Το νέο μέσο βρίσκεται στη θέση 2 όπου βρίσκεται και το στοιχείο αναζήτησης. Τέλος, ελέγχεται το μέσο και βρίσκεται ίσο με το στοιχείο αναζήτησης, οπότε η αναζήτηση διακόπτεται.

Ο κώδικας 6.19 παρέχει την υλοποίηση σε Java του αλγόριθμου δυαδικής αναζήτησης.

```
public static void binarySearch() { //Κώδικας 6.19
    int[] tbl = new int[]{1, 3, 4, 5, 7, 9, 11, 23, 34, 50};
    int idx = -1;
    final int schElement = 24;
    int from = 0, to = tbl.length - 1, mid = (to - from) / 2;
    while (to >= from) {
        if (tbl[mid] == schElement) {
            idx = mid;
            break;
        } else if (schElement < tbl[mid]) {
            to = mid - 1;
        } else {
            from = mid + 1;
        }
        mid = from + (to - from) / 2;
    }
    if (idx == -1) {
        System.out.println("Element " + schElement + " was not found");
    } else {
```

```
            System.out.println("Element " + schElement + " found at position " +
idx);
        }
    }
```

Κώδικας 6.19 *Υλοποίηση δυαδικής αναζήτησης*

Στον κώδικα 6.19 καθορίζουμε καταρχάς τον πίνακα στον οποίο θα εφαρμόσουμε τη δυαδική αναζήτηση. Στη συνέχεια ορίζουμε τις απαραιτητες βοηθητικές μεταβλητές idx, from, to και mid. Η μεταβλητή from αρχικοποιείται στο 0 και η to στο tbl.length-1. Έτσι, η αναζήτηση αρχικά αφορά όλο τον πίνακα. Αμέσως μετά, καθορίζουμε το στοιχείο αναζήτησης, schElement. Στη συνέχεια, η ροή εισέρχεται σε έναν βρόχο while που ελέγχεται από τη συνθήκη to>=from. Όσο αυτή η συνθήκη είναι αληθής, υπάρχει τμήμα του πίνακα που δεν έχει ελεγχθεί. Στη συνέχεια, αν η τιμή στο μέσο του πίνακα είναι ίση με την τιμή που αναζητούμε, το στοιχείο βρέθηκε, οπότε ενημερώνεται η idx και διακόπτεται η while, αλλιώς, αν το στοιχείο αναζήτησης είναι μικρότερο από την τιμή στο μέσο του πίνακα, ενημερώνεται η μεταβλητή to ώστε η αναζήτηση να περιοριστεί στο πρώτο μισό του πίνακα. Αν όμως το στοιχείο αναζήτησης είναι μεγαλύτερο από την τιμή στο μέσο του πίνακα, ενημερώνεται η from ώστε η αναζήτηση να συνεχιστεί στο δεύτερο μισό του πίνακα. Στο επόμενο βήμα υπολογίζεται το νέο μέσο. Τέλος, έξω από τον βρόχο while ενημερώνουμε με τα αποτελέσματα της αναζήτησης με την εμφάνιση κατάλληλων μηνυμάτων.

6.5.3 Ισότητα Πινάκων

Να αναπτύξετε συνάρτηση στην οποία να ορίσετε τους ακόλουθους πίνακες:

```
String[] tbl1 = new String[]{new String("John"), "Maria",
"Kely"};
String[] tbl2 = {"John", "Kely", "Maria"};
```

Στη συνέχεια να ελέγξετε αν είναι ίσοι μεταξύ τους με βάση τον ακόλουθο ορισμό ισότητας: Δύο πίνακες από Strings είναι ίσοι μεταξύ τους, εφόσον έχουν το ίδιο μέγεθος και κάθε στοιχείο του ενός είναι και στοιχείο του άλλου ανεξάρτητα από τη θέση που έχουν τα στοιχεία στους πίνακες.
Προσέξτε πως αυτός ο ορισμός ισότητας πινάκων διαφέρει από τον ορισμό στον οποίο βασίζονται οι συναρτήσεις equals της Array.

Λύση

```
public static void stringEquality() { //Κώδικας 6.20
    String[] tbl1 = new String[]{new String("John"), "Maria", "Kely"};
    String[] tbl2 = {"John", "Kely", "Maria"};
    boolean equal = tbl1.length == tbl2.length;
    for (int i = 0; i < tbl1.length && equal; i++) {
        boolean found = false;
        for (String cName : tbl2) {
            if (tbl1[i].equals(cName)) {
                found = true;
                break;
            }
        }
        equal = found;
    }
    System.out.println(equal);
}
```

Κώδικας 6.20 *Έλεγχος ισότητας πινάκων*

Στον κώδικα 6.20 ορίζουμε αρχικά τους πίνακες όπως δίνονται στην εκφώνηση. Στη συνέχεια αρχικοποιούμε τη μεταβλητή equal στην τιμή της έκφρασης tbl1.length == tbl2.length. Αν οι πίνακες δεν έχουν ίδιο μήκος, η equal γίνεται false. Στη συνέχεια η ροή δεν μπαίνει στον βρόχο for και απλά εμφανίζει την τιμή false. Αν όμως τα μήκη των πινάκων είναι ίσα μεταξύ τους, τότε η ροή εισέρχεται στον εξωτερικό βρόχο for. Εκεί

ελέγχεται κάθε στοιχείο του tbl1 αν υπάρχει και στον tbl2. Όσο ένα στοιχείο του tbl1 βρίσκεται και στον tbl2, η βοηθητική μεταβλητή found γίνεται true και αντίστοιχα ενημερώνεται και η equal. Αν ένα στοιχείο του tbl1 δεν βρεθεί στον tbl2, η found γίνεται false και διακόπτεται η εσωτερική for. Ακολούθως γίνεται false η equal η οποία θα διακόψει και την εξωτερική for.

6.5.4 Αντιστροφή πίνακα

Να γράψετε κώδικα ο οποίος με δεδομένο τον πίνακα int[] tbl = {1, 2, 3, 14, 5, 6, 7, 8, 9} παράγει έναν πίνακα που περιέχει τα ίδια στοιχεία με τον tbl σε αντεστραμμένη διάταξη, δηλαδή στη θέση 0 έχει το 9 και στη θέση 8 το 1.

Λύση

```
public static void reverse() { //Κώδικας 6.21
    int[] tbl = {1, 2, 3, 14, 5, 6, 7, 8, 9};
    int[] reversed = new int[tbl.length];
    for (int i = 0; i < tbl.length; i++) {
        reversed[tbl.length - 1 - i] = tbl[i];
    }
    System.out.print(Arrays.toString(reversed));
}
```

Κώδικας 6.21 *Παραγωγή πίνακα με αντεστραμμένη διάταξη στοιχείων*

Καταρχάς δημιουργούμε τον πίνακα reversed με ίδιο μήκος με τον tbl. Στη συνέχεια τοποθετούμε στη θέση tbl.length - 1 – i του reversed το στοιχείο που βρίσκεται στη θέση i του tbl.

6.5.5 Συνένωση πινάκων

Να γράψετε κώδικα που με δεδομένους τους πίνακες int[] t1={1,2,3,4,5}, t2={6,7,8,9} παράγει έναν πίνακα που συνενώνει τους t1 και t2.

Λύση

```
public static void merge() { //Κώδικας 6.22
    int[] t1 = {1, 2, 3, 4, 5}, t2 = {6, 7, 8, 9};
    int[] rVal = new int[t1.length + t2.length];
    System.arraycopy(t1, 0, rVal, 0, t1.length);
    System.arraycopy(t2, 0, rVal, t1.length, t2.length);
    System.out.println(Arrays.toString(rVal));
}
```

Κώδικας 6.22 *Συνένωση πινάκων*

Αφού δηλώσαμε τους πίνακες t1 και t2, δημιουργούμε τον πίνακα rVal με μήκος ικανό να φιλοξενήσει τα δεδομένα των t1 και t2. Στη συνέχεια, με την arraycopy μεταφέρουμε τα δεδομένα από τους t1 και t2. Τέλος, τυπώνουμε τον rVal ώστε να δείξουμε πως η συνένωση έγινε σωστά.

6.5.6 Εφαρμογή της arraycopy

Να γράψετε κώδικα που καταρχάς ορίζει τον πίνακα

```
char[][] copyFrom = {
      {'d', 'e'},
      {'c', 'a', 'f', 'f', 'e', 'i', 'n'},
      {'a', 't', 'e', 'd'}
};
```

Στη συνέχεια, ορίστε έναν πίνακα, copyTo, ικανών διαστάσεων ώστε να μπορεί να φιλοξενήσει αντίγραφο του copyFrom. Στη θέση 2 του copyTo τοποθετήστε τους χαρακτήρες 'l', 'i', 'k', 'e'. Αντιγράψτε από τον copyFrom στον copyTo τους πίνακες στις θέσεις 0 και 1. Εμφανίστε τον copyTwo.

Λύση

```
public static void copy() { ////Κώδικας 6.23
    char[][] copyFrom = {
        {'d', 'e'},
        {'c', 'a', 'f', 'f', 'e', 'i', 'n'},
        {'a', 't', 'e', 'd'}
    };
    char[][] copyTo = new char[3][];
    copyTo[2] = new char[]{'l', 'i', 'k', 'e'};
    System.arraycopy(copyFrom, 1, copyTo, 0, 2);
    for (char[] aElement : copyTo) {
        System.out.println(Arrays.toString(aElement));
    }
}
```

Κώδικας 6.23 *Λύση της άσκησης 6*

6.5.7 Μεταβολή μεγέθους

Να υλοποιήσετε συνάρτηση στην οποία να δηλώνετε έναν μονοδιάστατο πίνακα ακέραιων 6 θέσεων, έστω t. Στη συνέχεια καταχωρήστε τους ακεραίους 1 έως 6 στον t. Ορίστε επίσης μια ακέραια μεταβλητή, έστω newSize. Μεταβάλλεται το μήκος του t σε newSize. Φροντίστε ώστε ο κώδικάς σας να λειτουργεί σωστά είτε το newSize είναι μεγαλύτερο από το 6 είτε μικρότερο. Τυπώστε τον t ώστε να διαπιστώσετε πως έχει το αναμενόμενο μήκος.

Λύση

```
1  static void resize() { //Κώδικας 6.24
2      int[] t={1,2,3,4,5,6};
3      int newSize=4;
4      int[] tmp=new int[newSize];
5      System.arraycopy(t, 0, tmp, 0, t.length<newSize?t.length:newSize);
6      t=tmp;
7      System.out.println(Arrays.toString(t));
8  }
```

Κώδικας 6.24 *Μεταβολή μεγέθους πίνακα*

Ορίζουμε τον t και την newSize σύμφωνα με την εκφώνηση. Στη συνέχεια δημιουργούμε έναν άλλο πίνακα με μήκος newSize. Αμέσως μετά αντιγράφουμε δεδομένα από τον t στον tmp. Προσοχή σε αυτό το σημείο. Αν το newSize είναι μεγαλύτερο ή ίσο από το t.length, τότε μπορούν να αντιγραφούν όλα τα δεδομένα του t. Αν όμως είναι μικρότερο, τότε θα πρέπει να αντιγραφούν μόνο όσα χωρούν. Στον κώδικα 6.24 ο έλεγχος αυτός γίνεται με τη βοήθεια του τριαδικού τελεστή στη γραμμή 5. Στη γραμμή 6 εκχωρούμε την αναφορά tmp στην t. Στη συνέχεια η t δείχνει σε πίνακα μεγέθους newSize.

6.6 Ασκήσεις προς λύση

1. Να γράψετε κώδικα στον οποίο να ορίζετε τους δισδιάστατους πίνακες ακέραιων 3x4, t1 και t2. Στη συνέχεια να δημιουργήσετε ένα πίνακα sum τέτοιον ώστε sum[i][j]=t1[i][j]+t2[i][j]. Εμφανίστε τον sum και ελέγξτε την ορθότητα του κώδικα.

2. Να γράψετε κώδικα που υπολογίζει ποια γραμμή δισδιάστατου πίνακα πραγματικών έχει το μεγαλύτερο άθροισμα στοιχείων.

3. Να γράψετε κώδικα που υπολογίζει ποια στήλη δισδιάστατου πίνακα πραγματικών έχει το μεγαλύτερο άθροισμα στοιχείων.

4. Να γράψετε κώδικα που αρχικοποιεί πίνακα ακέραιων 10x5 με τυχαίες τιμές από 0 έως 8. Τα δεδομένα σε κάθε γραμμή του πίνακα αναπαριστούν τις ώρες που εργάστηκε ένας εργαζόμενος κάθε μια από τις 5 εργάσιμες ημέρες μιας εβδομάδας. Υπολογίστε το σύνολο των ωρών που εργάστηκε κάθε εργαζόμενος κατά τη διάρκεια της εβδομάδας. Εμφανίστε τα σύνολα ταξινομημένα κατά φθίνουσα σειρά.

5. Να γράψετε κώδικα που αρχικοποιεί έναν δισδιάστατο πίνακα ακέραιων σε τυχαίες τιμές 0 και 1. Να υπολογίσετε πόσα 0 και πόσα 1 υπάρχουν σε κάθε γραμμή του πίνακα και πόσα σε όλον τον πίνακα.

6. Σε έναν δισδιάστατο πίνακα από String καταχωρήστε σε κάθε γραμμή του ένα κράτος της Ευρώπης με την πρωτεύουσά του. Στη συνέχεια υποβάλλετε 10 ερωτήσεις στον χρήστη και αξιολογήστε τις απαντήσεις του. Κάθε ερώτηση μπορεί να δίνει την ονομασία του κράτους και να ζητά την πρωτεύουσα ή να δίνει την πρωτεύουσα και να ζητά το κράτος.

7. Να αναπτυχθεί συνάρτηση που λαμβάνει ως παράμετρο έναν πίνακα ακέραιων και επιστρέφει 1 αν ο πίνακας είναι ταξινομημένος κατ’ αύξουσα σειρά, -1 εν είναι ταξινομημένος κατά φθίνουσα σειρά και 0 αν δεν είναι ταξινομημένος.

8. Να δηλώσετε έναν πίνακα, a, από συμβολοσειρές. Να εκχωρήσετε μια συμβολοσειρά σε κάθε θέση του πίνακα. Στη συνέχεια, να δημιουργήσετε ένα αντίγραφο του a τέτοιο ώστε κάθε αλλαγή σε μια συμβολοσειρά του a να μην επηρεάζει την αντίστοιχη συμβολοσειρά στο αντίγραφο.

Βιβλιογραφία

[1] C. Horstmann, Big Java: Early Objects, 7th Edition | Wiley, 7th ed. Laurie Rosatone, 2013. Accessed: Sep. 14, 2021. [Online]. Available: https://www.wiley.com/en-us/Big+Java%3A+Early+Objects%2C+7th+Edition-p-9781119499091

[2] D. Liang, Εισαγωγή στον Προγραμματισμό Java, 10η. Θεσσαλονίκη: Τζιόλας.

Κεφάλαιο 7

Σύνοψη
Σε αυτήν την ενότητα γίνεται εισαγωγή στις στατικές συναρτήσεις. Παρουσιάζονται οι τυπικές και πραγματικές παράμετροι καθώς και η τιμή επιστροφής. Εξηγείται η διαφορά μεταξύ παραμέτρων τιμής και αναφοράς. Αναλύεται η λίστα παραμέτρων μεταβλητού μήκους και οι παράμετροι της main. Επίσης, συζητούνται τρόποι αξιοποίησης στατικών μεταβλητών ενώ γίνονται οι απαραίτητες επισημάνσεις για την πρόκληση τυχόν απροσδόκητων λαθών κατά την κλήση συναρτήσεων. Επιπλέον, παρουσιάζονται μερικές χρήσιμες προκαθορισμένες συναρτήσεις, συζητούνται οι προσδιοριστές προσπέλασης και αναλύεται η χρήση των πακέτων.

Προαπαιτούμενη γνώση
Οι θεμελιώδεις τύποι και βασική γνώση του τύπου String. Βασική είσοδος και έξοδος. Η λειτουργία των τελεστών. Οι δομές επιλογής και οι επαναληπτικές δομές. Η διαχείριση των πινάκων.

Λέξεις κλειδιά
Στατικές συναρτήσεις, στατικές μεταβλητές, Υπερφόρτωση, τυπική παράμετρος, πραγματική παράμετρος, παράμετρος τιμής, παράμετρος αναφοράς, λίστα παραμέτρων μεταβλητού μήκους, τιμή επιστροφής.

7. Στατικές Συναρτήσεις και Μεταβλητές

Είναι συχνό φαινόμενο το ίδιο τμήμα κώδικα να είναι αναγκαίο σε διάφορα σημεία ενός προγράμματος. Ένας απλοϊκός τρόπος για να αντιμετωπιστεί η αναγκαιότητα αυτή είναι να επαναλάβουμε το ίδιο τμήμα όπου και όσες φορές αυτό χρειάζεται. Όμως, η προσέγγιση αυτή παρουσιάζει σοβαρά μειονεκτήματα. Καταρχάς, καθιστά τη συντήρηση του κώδικα πολύ δύσκολη. Αν χρειαστεί να κάνω μια βελτίωση ή οποιαδήποτε άλλη μεταβολή στο τμήμα του κώδικα που επαναλαμβάνεται, τότε θα πρέπει να βρω όλα τα σημεία στα οποία ενσωματώνεται και να τα μεταβάλω ένα προς ένα. Φανταστείτε τώρα πόσο δυσχεραίνεται η συντήρηση του κώδικα αν στο πρόγραμμά σας έχετε πολλά τμήματα κώδικα που το καθένα χρειάζεται σε πολλά σημεία, κάτι που είναι πολύ συνηθισμένο στα σύγχρονα προγράμματα.

Ένα άλλο θέμα μάλλον σημαντικότερο και από τη δυσκολία στη συντήρηση είναι η έλλειψη αφαιρετικότητας (abstraction). Ας υποθέσουμε πως έχουμε ένα τμήμα κώδικα που υπολογίζει τη δύναμη ενός αριθμού υψωμένου σε μια δύναμη. Έστω κατά την ανάπτυξη της εφαρμογής μας, χρειαζόμαστε αυτόν τον υπολογισμό. Πού είναι το τμήμα του κώδικα που τον κάνει; Θα πρέπει να ψάξουμε. Έστω ότι το βρήκαμε. Ναι, όμως εκείνο το τμήμα υπολογίζει την τιμή της μεταβλητής x υψωμένης στην τιμή της μεταβλητής k. Σε ένα άλλο σημείο χρειαζόμαστε τον ίδιο υπολογισμό αλλά για διαφορετικές μεταβλητές. Πρέπει λοιπόν να συντηρούμε σε πολλά σημεία όχι απλώς τον ίδιο κώδικα αλλά περίπου τον ίδιο. Οι μεταβολές όμως από σημείο σε σημείο ενδέχεται να εισάγουν λάθη.

Σίγουρα θα ήταν βολικό να κατασκευάζαμε ένα υποπρόγραμμα που θα είχε κατάλληλη ονομασία, π.χ. power ή calcPower έτσι ώστε όποτε χρειαζόμασταν να υπολογίσουμε τη δύναμη ενός αριθμού, να το καλούσαμε να έκανε τους υπολογισμούς και να μας ενημέρωνε με το αποτέλεσμα. Ένα τέτοιο υποπρόγραμμα βέβαια δεν θα έπρεπε να υπολογίζει κάθε φορά τη δύναμη του ίδιου αριθμού υψωμένου στον ίδιο εκθέτη. Επομένως θα έπρεπε να υπάρχει τρόπος κάθε φορά που το καλούμε να το ενημερώνουμε σχετικά με τη βάση και τον εκθέτη του απαιτούμενου υπολογισμού.

Για καλή μας τύχη, τέτοιου είδους υποπρογράμματα υποστηρίζονται στην Java και ονομάζονται συναρτήσεις. Οι παράμετροι των συναρτήσεων είναι ο μηχανισμός μέσω του οποίου δίνουμε στη συνάρτηση τα δεδομένα στα οποία θα επενεργήσει ενώ ο κύριος τρόπος που η συνάρτηση μας επιστρέφει την απάντησή της είναι η τιμή επιστροφής της συνάρτησης.

Με αυτόν τον τρόπο επιτυγχάνουμε σημαντική αφαιρετικότητα στα προγράμματά μας. Δεν είναι αναγκαίο να γνωρίζουμε τις λεπτομέρειες του κώδικα μιας συνάρτησης για να τη χρησιμοποιήσουμε. Αρκεί να γνωρίζουμε τι κάνει (και όχι πώς το κάνει), το όνομά της, τις παραμέτρους που δέχεται και τι αποτελέσματα επιστρέφει. Έτσι είναι εύκολο να χρησιμοποιηθούν συναρτήσεις και μάλιστα από ένα ευρύτερο κοινό και όχι μόνο από αυτούς που τις έχουν υλοποιήσει.

Πέραν της τιμής επιστροφής της, μια συνάρτηση είναι δυνατό να παράγει αποτελέσματα και με άλλους τρόπους. Για παράδειγμα, μία συνάρτηση εισάγει σε έναν πίνακα μια σειρά από τιμές και επιστρέφει τον

αριθμό των τιμών που εισήχθησαν. Τα αποτελέσματα που παράγει η συνάρτηση πέραν της τιμής επιστροφής της είναι γνωστά ως παράπλευρα αποτελέσματα (side effects). Η Java, όπως θα δούμε αμέσως παρακάτω σε αυτήν την ενότητα, υποστηρίζει συναρτήσεις που παράγουν μόνο παράπλευρα αποτελέσματα χωρίς να επιστρέφουν κάποια τιμή. Άλλες γλώσσες διαφοροποιούν τα υποπρογράμματα που παράγουν μόνο παράπλευρα αποτελέσματα από αυτά που επιστρέφουν κάποια τιμή. Για παράδειγμα, η Pascal ονομάζει τα υποπρογράμματα που δεν επιστρέφουν κάποια τιμή διαδικασίες (procedures), ενώ διατηρεί την ονομασία συναρτήσεις (functions) για τα υποπρογράμματα που επιστρέφουν τιμή.

Επάνω στη δυνατότητα να αναπτύσσουμε συναρτήσεις βασίζεται η δυνατότητα να οργανώνουμε τους πηγαίους κώδικες με δομημένο τρόπο. Δημιουργείται έτσι το μοντέλο του δομημένου προγραμματισμού (structured programming). Πρόκειται για ένα σημαντικό μοντέλο προγραμματισμού που διευκολύνει την ανάπτυξη και συντήρηση των προγραμμάτων, ενώ αποτελεί συστατικό στοιχείο του Αντικειμενοστρεφούς μοντέλου.

Η Java υποστηρίζει δύο ειδών συναρτήσεις, τις στατικές (static) και τις μη στατικές (non static). Το κύριο θέμα αυτού του κεφαλαίου είναι οι στατικές συναρτήσεις. Οι μη στατικές συναρτήσεις και μεταβλητές παρουσιάζονται αργότερα σε αυτό το εγχειρίδιο κατά την εισαγωγή στο Αντικειμενοστρεφές μοντέλο, στην ενότητα 11. Εδώ μαζί με την παρουσίαση των στατικών συναρτήσεων, παρουσιάζουμε και τις στατικές μεταβλητές της κλάσης.

7.1 Στατικές Συναρτήσεις

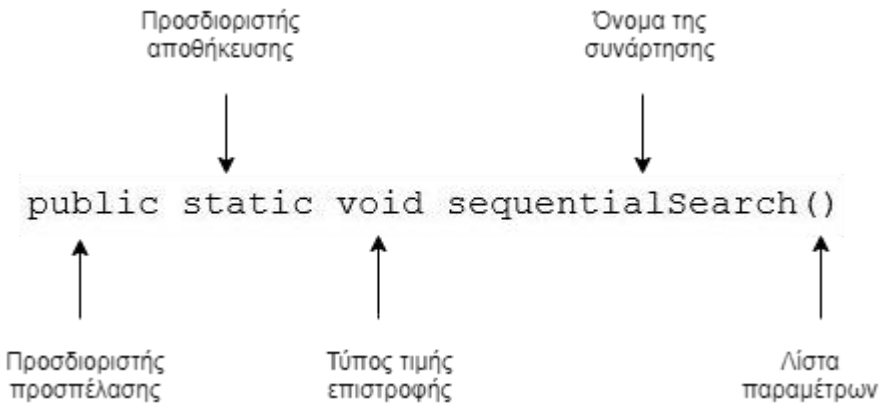
Έχουμε ήδη χρησιμοποιήσει στατικές συναρτήσεις στην πολύ απλή τους μορφή. Ας εξετάσουμε καταρχάς τη συνάρτηση sequentialSearch του κώδικα 6.12. Το τμήμα

```
public static void sequentialSearch()
```

ονομάζεται επικεφαλίδα της συνάρτησης (function header). Το μπλοκ που ακολουθεί την επικεφαλίδα ονομάζεται σώμα της συνάρτησης (function body). Η επικεφαλίδα της συνάρτησης ονομάζεται και διεπαφή της συνάρτησης (function interface), γιατί είναι το τμήμα τις λεπτομέρειες του οποίου είναι αναγκαίο να γνωρίζει ο χρήστης της συνάρτησης για να είναι σε θέση να την καλεί και να λαμβάνει την επιστροφή της συνάρτησης. Η επικεφαλίδα μαζί με το σώμα της συνάρτησης αποτελούν τον ορισμό της συνάρτησης (function definition). Ο ορισμός της συνάρτησης από μόνος του δεν παράγει κάποιο αποτέλεσμα σε ένα πρόγραμμα. Το αποτέλεσμα το παράγει η κλίση της συνάρτησης. Για παράδειγμα, η sequentialSearch χρειάζεται μια ακέραια μεταβλητή που ονομάζει schElement. Μνήμη για αυτήν τη μεταβλητή δεσμεύεται μόνον κατά την κλίση της συνάρτησης.

Στην αρχή της επικεφαλίδας έχουμε έναν προσδιοριστή προσπέλασης, στη συγκεκριμένη περίπτωση τον προσδιοριστή public. Για τους προσδιοριστές προσπέλασης έχουμε ήδη μιλήσει στην ενότητα 2.3. Στη συνέχεια ακολουθεί ο προσδιοριστής αποθήκευσης (class storage specifier), static που καθορίζει πως πρόκειται για στατική συνάρτηση. Μία στατική συνάρτηση μπορεί να κληθεί μέσω της κλάσης στην οποία ορίζεται όπως φαίνεται στον κώδικα 2.6.

Στη συνέχεια ακολουθεί η δεσμευμένη λέξη void που ορίζει πως η συνάρτηση αυτή δεν έχει τιμή επιστροφής. Μετά ακολουθεί το όνομα της συνάρτησης που είναι sequentialSearch και τέλος πριν το σώμα της συνάρτησης παρεμβάλλεται η λίστα παραμέτρων που στο παράδειγμά μας είναι κενή. Στο σχήμα 7.1 παρουσιάζονται τα συστατικά μέρη της διεπαφής της συνάρτησης.



Σχήμα 7.1 *Η διεπαφή της συνάρτησης*

Συνεχίζοντας την εξέταση της sequentialSearch παρατηρούμε πως αυτή ψάχνει να εντοπίσει μια συγκεκριμένη τιμή σε έναν συγκεκριμένο πίνακα. Μάλιστα τιμή αναζήτησης και πίνακας ορίζονται μέσα στην ίδια τη συνάρτηση. Αυτό σημαίνει πως η συνάρτηση αυτή δεν μπορεί να χρησιμοποιηθεί για κανέναν άλλο πίνακα αλλά ούτε και για να ψάξει άλλη τιμή έστω και στον ίδιο πίνακα. Ο αλγόριθμος όμως της σειριακής αναζήτησης είναι ίδιος για οποιονδήποτε μονοδιάστατο πίνακα και για οποιαδήποτε τιμή αναζήτησης. Επομένως, θα ήταν χρήσιμο να απομονώσουμε τον αλγόριθμο της σειριακής αναζήτησης κατά τρόπο ώστε να μπορούμε να τον εφαρμόζουμε σε οποιονδήποτε πίνακα και για οποιαδήποτε τιμή. Λύση σε αυτό το ζήτημα παρέχει η λίστα παραμέτρων της συνάρτησης. Η λίστα παραμέτρων δίνει τη δυνατότητα σε αυτόν που καλεί τη συνάρτηση να καθορίζει τόσο τον πίνακα όσο και την τιμή στα οποία επενεργεί η συνάρτηση. Όμως για να επιτευχθεί αυτό θα πρέπει κατά τον ορισμό της συνάρτησης να καθορίσουμε τους τύπους των παραμέτρων, όπως δείχνει παρακάτω η νέα διεπαφή της συνάρτησης.

```
public static void sequentialSearch(int[] array, int
schElement)
```

Η νέα διεπαφή επιβάλλει ώστε κατά την κλήση της συνάρτησης να δίνεται ένας πίνακας ακέραιων και μια ακέραια τιμή. Είναι προφανές πως κάποιες μεταβολές θα πρέπει να γίνουν στο σώμα της sequentialSearch. Ας δούμε όμως πρώτα πώς καλείται μια συνάρτηση με τη νέα διεπαφή της sequentialSearch.

```
int[] t1 = {3, 2, 5, 3, 6, 9};
int[] t2 = {6, 4, 5, 2, 1, 9, 9};
int sItem = 5;
sequentialSearch(t1, 9);
sequentialSearch(t2, sItem);
```

Κώδικας 7.1 *Κλήση της sequentialSearch με παραμέτρους, χωρίς τιμή επιστροφής*

Στον κώδικα 7.1 βλέπουμε δύο κλήσεις της sequentialSearch. Κατά την πρώτη κλήση η συνάρτηση αναζητά στον πίνακα t1 την τιμή 9 και στη δεύτερη κλήση αναζητά στον πίνακα t2 την τιμή της μεταβλητής sItem, δηλαδή την τιμή 5.

Ας σημειωθεί πως οι παράμετροι στον ορισμό της συνάρτησης ονομάζονται τυπικές παράμετροι (formal parameters), ενώ οι παράμετροι που δίνει ο χρήστης της συνάρτησης κατά την κλήση της ονομάζονται πραγματικές παράμετροι (actual parameters).

Το πρόβλημα στον σχεδιασμό της συνάρτησης όπως τον έχουμε τώρα είναι πως η συνάρτηση δεν επιστρέφει στον κώδικα που την καλεί το αποτέλεσμα της αναζήτησης. Αυτό όμως είναι αναγκαίο καθώς στο αποτέλεσμα της αναζήτησης, ο κώδικας που την καλεί θα βασίσει την περαιτέρω ροή του. Για παράδειγμα, ενδέχεται αν η τιμή βρεθεί, ο χρήστης της συνάρτησης να επιθυμεί να αυξήσει την τιμή στον πίνακα κατά 1. Αν πάλι η τιμή δεν βρεθεί, μπορεί να μην χρειάζεται καμία περαιτέρω ενέργεια για τον συγκεκριμένο πίνακα.

Το πρόβλημα αυτό λύνεται με την τιμή επιστροφής της συνάρτησης. Στο συγκεκριμένο παράδειγμα μια καλή τιμή επιστροφής είναι τύπου int. Στην περίπτωση που θα βρεθεί η τιμή αναζήτησης, ο ακέραιος που θα επιστραφεί θα αναπαριστά τη θέση της τιμής αναζήτησης στον πίνακα. Αν πάλι η τιμή δεν βρεθεί, μπορεί να επιστραφεί η τιμή -1. Καθώς δεν υπάρχει θέση του πίνακα με αρνητική τιμή, η τιμή -1 μπορεί να

αναπαραστήσει με ασφάλεια την αποτυχία της αναζήτησης. Προσέξτε εδώ πως αν διαλέξουμε να αναπαραστήσουμε την αποτυχία αναζήτησης με μη αρνητικό αριθμό, π.χ. με 0, τότε έχουμε κακό σχεδιασμό της συνάρτησης. Αυτό γιατί η συνάρτηση θα επιστρέφει 0, τόσο στην περίπτωση που το στοιχείο αναζήτησης δεν βρεθεί, όσο και στην περίπτωση που βρεθεί στη θέση 0 του πίνακα.

Επομένως, η διεπαφή της συνάρτησης τροποποιείται όπως δείχνει ο ακόλουθος κώδικας:

```
public static int sequentialSearch(int[] array, int
schElement)
```

Η τιμή επιστροφής έγινε int από void που ήταν στην προηγούμενη έκδοση. Τώρα η συνάρτηση μπορεί να κληθεί όπως δείχνει ο κώδικας 7.2:

```
static void callSeqSearch() { //Κώδικας 7.2
    int[] t1 = {3, 2, 5, 3, 6, 9};
    int sItem = 5;
    int idx = sequentialSearch(t1, sItem);
    if (idx > -1) {
        t1[idx]++;
    }
}
```

Κώδικας 7.2 *Κλήση της sequentialSearch με παραμέτρους και τιμή επιστροφής*

Στον κώδικα 7.2, η κλήση της συνάρτησης επιστρέφει τη θέση του πίνακα στην οποία βρέθηκε το sItem ή -1. Αν επιστρέψει -1, καμία ενέργεια επάνω στον πίνακα δεν γίνεται. Αν όμως επιστρέψει μη αρνητική τιμή, τότε προσαυξάνεται η τιμή στη θέση που επιστράφηκε κατά 1.

Ακόμη και αν κάνουμε τις αλλαγές που συζητήσαμε μέχρι τώρα, παραμένει ένα σοβαρό μειονέκτημα στη συνάρτηση. Αυτό συνίσταται στο γεγονός ότι η συνάρτηση εμφανίζει τα αποτελέσματα της αναζήτησης. Προσέξτε, το μειονέκτημα δεν είναι ότι επιστρέφει στον κώδικα που την καλεί τα αποτελέσματα της αναζήτησης αλλά ότι τα εμφανίζει χρησιμοποιώντας μάλιστα την println. Αυτό το μειονέκτημα πολλές φορές γίνεται δύσκολα κατανοητό στους αρχάριους προγραμματιστές. Πράγματι, αν κάνω έναν κώδικα που ψάχνει σε συγκεκριμένο πίνακα για να βρει μια συγκεκριμένη τιμή και το μόνο που θέλω είναι να μάθω αν η τιμή βρέθηκε και που, τότε δεν είναι πρόβλημα αν ο κώδικας εμφανίζει το αποτέλεσμα της αναζήτησης. Μια συνάρτηση όμως μπορεί να κληθεί σε ένα σημείο που θέλω να εμφανίσω το αποτέλεσμά της αλλά μπορεί να κληθεί και σε πολλά σημεία που δεν θέλω να το εμφανίζω αλλά απλώς το χρειάζομαι μέσα στον κώδικά μου. Για παράδειγμα, μπορεί την sequentialSearch να την καλέσω σε μια παραθυρική εφαρμογή. Σε αυτήν την περίπτωση η έξοδος με την println που είναι κατάλληλη μόνο για περιβάλλοντα γραμμής εντολών (command line) προφανώς θα μου καταστρέψει τη διεπαφή χρήστη (user interface).

Ο κώδικας 7.3 δίνει τον ορισμό της sequentialSearch που ενσωματώνει όλες τις παρατηρήσεις μέχρι εδώ.

```
public static int sequentialSearch(int[] array, int sItem) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == sItem) {
            return i;
        }
    }
    return -1;
}
```

Κώδικας 7.3 *Συνάρτηση σειριακής αναζήτησης σε μονοδιάστατο πίνακα ακέραιων*

Η sequentialSearch σε αυτή τη μορφή μπορεί να κληθεί από οποιονδήποτε κώδικα για να αναζητήσει σε οποιονδήποτε μονοδιάστατο πίνακα ακέραιων οποιαδήποτε ακέραια τιμή. Προσέξτε πως η επιστροφή τιμής από τη συνάρτηση γίνεται με χρήση της δεσμευμένης λέξης return. Η λέξη return συντάσσεται με έκφραση που ο τύπος της είναι υποχρεωτικά ίδιος ή συμβατός με τον τύπο της συνάρτησης. Η return προκαλεί άμεση έξοδο από τη συνάρτηση. Σε συναρτήσεις τύπου void, η return συντάσσεται μόνη, ακολουθούμενη απλώς

από τον χαρακτήρα ελληνικό ερωτηματικό (semicolon). Σε void συναρτήσεις, η return δεν είναι υποχρεωτική. Ωστόσο, σε κάποιες περιπτώσεις είναι χρήσιμη.

Οι παράμετροι των συναρτήσεων στην Java είναι παράμετροι θέσης (positional parameters). Αυτό σημαίνει πως κατά την κλήση συνάρτησης, οι πραγματικές παράμετροι αντιστοιχίζονται με τις τυπικές παραμέτρους ανάλογα με τη θέση που διατηρεί κάθε μία στην λίστα παραμέτρων.

Τέλος να επισημάνουμε πως οι συναρτήσεις στα πλαίσια του αντικειμενοστρεφούς προγραμματισμού ονομάζονται και μέθοδοι (methods).

7.2 Υπερφόρτωση Συναρτήσεων

Πάντως, είναι πιθανό να χρειάζεται να κάνουμε αναζήτηση τιμών και σε πίνακες που δεν είναι πίνακες ακέραιων. Ας υποθέσουμε πως χρειάζεται να κάνουμε αναζήτηση σε έναν πίνακα πραγματικών. Σε αυτήν την περίπτωση μπορούμε να υλοποιήσουμε μια άλλη συνάρτηση με παραμέτρους τύπου double[] και double. Το θέμα είναι πως παρότι αλλάζουν οι τύποι των δεδομένων, η λειτουργία παραμένει ίδια. Δεν θέλουμε για την ίδια λειτουργία να έχουμε συναρτήσεις με διαφορετικό όνομα. Παρόλα αυτά, για κάποιες γλώσσες προγραμματισμού, η μόνη λύση είναι να κάνουμε μια συνάρτηση που έχει διαφορετικό όνομα από τη συνάρτηση που ψάχνει σε πίνακες ακέραιων. Αυτό όμως δεν είναι βολικό γιατί δεν θέλουμε να αναγκαζόμαστε να εφευρίσκουμε και να θυμόμαστε διαφορετικά ονόματα για την ίδια λειτουργία. Ευτυχώς, η Java όπως και οι περισσότερες σύγχρονες γλώσσες προγραμματισμού, επιλύουν αυτό το θέμα με την υπερφόρτωση συναρτήσεων (function overloading).

Με την υπερφόρτωση συναρτήσεων μπορούμε να ορίσουμε συναρτήσεις με το ίδιο όνομα αρκεί να διαφέρει η ταυτότητά τους (function signature). Η ταυτότητα μιας συνάρτησης αποτελείται από το όνομα της συνάρτησης και τη λίστα παραμέτρων. Προσέξτε πως η τιμή επιστροφής δεν συμπεριλαμβάνεται στην ταυτότητα. Επομένως, μπορούμε να κάνουμε συναρτήσεις με το ίδιο όνομα, αρκεί να διαφέρει η λίστα παραμέτρων. Ποια από δύο συναρτήσεις που έχουν το ίδιο όνομα θα κληθεί είναι μια απόφαση που λαμβάνει ο μεταγλωττιστής ανάλογα με τις πραγματικές παραμέτρους. Στον κώδικα 7.4 δίνουμε παράδειγμα ορισμού και χρήσης υπερφορτωμένων συναρτήσεων.

```
static void f(int i) { //Κώδικας 7.4
      System.out.println("f(int)");
  }

  static void f(double d) { //Κώδικας 7.4
      System.out.println("f(double)");
  }

  public static void main(String[] args) { //Κώδικας 7.4
      f(1);
      f(1d);
  }
```

Κώδικας 7.4 *Ορισμός και χρήση υπερφορτωμένων συναρτήσεων*

Στον κώδικα 7.4 και οι δύο συναρτήσεις που ορίζουμε ονομάζονται f. Η μία όμως δέχεται μια ακέραια παράμετρο και η άλλη μια πραγματική παράμετρο. Στην main καλούμε την f την πρώτη φορά με ακέραια παράμετρο και τη δεύτερη με πραγματική. Η έξοδος του κώδικα είναι:

```
f(int)
f(double)
```

Η έξοδος αυτή μας δείχνει πως την πρώτη φορά κλήθηκε η f που λαμβάνει την ακέραια παράμετρο και τη δεύτερη φορά κλήθηκε η f που λαμβάνει την πραγματική παράμετρο.
Έτσι, αν θέλουμε να έχουμε και μια sequentialSearch ώστε να ψάχνει και σε πραγματικούς πίνακες δεν έχουμε παρά να την υπερφορτώσουμε όπως δείχνει ο κώδικας 7.5.

```
public static int sequentialSearch(double[] array, double sItem) {
    for (int i = 0; i < array.length; i++) {
```

```
            if (approximateEquals(array[i], sItem, 0.0001)) {
                return i;
            }
        }
        return -1;
    }
```

Κώδικας 7.5 *Σειριακή αναζήτηση σε πίνακες πραγματικών*

7.3 Παράμετροι

Έχουμε ήδη συζητήσει σχετικά με τις μεταβλητές αναφοράς και τις μεταβλητές τιμής. Η συμπεριφορά αυτών των δύο τύπων μεταβλητών είναι διαφορετική όταν περνάνε σαν παράμετροι σε συναρτήσεις. Σε αυτήν την ενότητα εξετάζουμε πρώτα τις μεταβλητές τιμής ως παραμέτρους και στη συνέχεια τις μεταβλητές αναφοράς.

7.3.1 Οι μεταβλητές τιμής ως παράμετροι

Ο κώδικας 7.6 παρουσιάζει δύο συναρτήσεις, την increment και την main. Μελετήστε τον κώδικα και επιχειρήστε να υποθέσετε την έξοδό του.

```
static void increment(int i) {
    i++;
    System.out.println(i);
}

public static void main(String[] args) {
    int i = 0;
    increment(i);
    System.out.println(i);
}
```

Κώδικας 7.6 *Παράδειγμα παραμέτρων τιμής*

Η συνάρτηση increment λαμβάνει ως παράμετρο έναν ακέραιο. Το αναγνωριστικό της παραμέτρου είναι i. Στην πρώτη γραμμή αυξάνει την τιμή της παραμέτρου κατά 1 και στη συνέχεια την τυπώνει.

Η main ορίζει μια ακέραια μεταβλητή, επίσης με το αναγνωριστικό i και την αρχικοποιεί στο 0. Στη συνέχεια καλεί την increment με πραγματική παράμετρο το i. Μετά το πέρας της increment, η main τυπώνει την i.

Πρώτη σημαντική παρατήρηση είναι ότι η i της increment και η i της main δεν ταυτίζονται. Μπορεί να είναι και οι δύο τύπου int και να έχουν το ίδιο αναγνωριστικό, ωστόσο πρόκειται για διαφορετικές μεταβλητές. Εξάλλου θα μπορούσαν να έχουν διαφορετικά αναγνωριστικά. Αν μεταβάλλετε το αναγνωριστικό στην main, π.χ. το κάνετε j, θα διαπιστώσετε πως δεν θα έχετε καμία διαφορά στο αποτέλεσμα του κώδικα.

Αν τρέξουμε τον κώδικα, θα διαπιστώσουμε πως η μεν increment τυπώνει 1, η δε main τυπώνει 0. Αυτή η συμπεριφορά εξηγείται ως εξής: Η main δεσμεύει χώρο στη μνήμη κατάλληλο για τη μεταβλητή i, δηλαδή χώρο κατάλληλο για έναν ακέραιο. Σε αυτήν τη θέση μνήμης εκχωρείται η τιμή 0. Όταν καλείται η increment, ένα αντίγραφο της πραγματικής παραμέτρου περνάει στην τυπική παράμετρο. Στη συνέχεια η i της increment λειτουργεί σαν τοπική μεταβλητή. Η increment δηλαδή δεσμεύει επίσης χώρο για έναν ακέραιο στον οποίο αντιγράφεται η τιμή της πραγματικής παραμέτρου. Από εκεί και μετά, η increment επενεργεί επάνω στη δική της μεταβλητή i που είναι εντελώς ανεξάρτητη από την i της main. Με το τέλος εκτέλεσης της increment, ο έλεγχος επανέρχεται στην main. Η μεταβλητή i της main έχει διατηρήσει την τιμή της και έτσι εμφανίζεται το 0.

Οι μεταβλητές όλων των θεμελιωδών τύπων στην Java είναι μεταβλητές τιμής και περνούν στις συναρτήσεις σαν παράμετροι τιμής.

7.3.2 Οι μεταβλητές αναφοράς ως παράμετροι

Είπαμε παραπάνω πως οι θεμελιώδεις τύποι περνάνε στις συναρτήσεις σαν παράμετροι τιμής. Το ίδιο ισχύει και για τις μεταβλητές αναφοράς. Ωστόσο η συμπεριφορά είναι διαφορετική. Μελετήστε τον κώδικα 7.7. Ποια είναι κατά την γνώμη σας η έξοδός του;

```
static void increment(int[] store) { //Κώδικας 7.7
    for (int i = 0; i < store.length; i++) {
        store[i]++;
    }
    System.out.println(Arrays.toString(store));
}

public static void main(String[] args) {
    int[] s = {1, 2, 3};
    increment(s);
    System.out.println(Arrays.toString(s));
}
```

Κώδικας 7.7 *Η μεταβλητή αναφοράς ως παράμετρος*

Η έξοδος του κώδικα 7.7 είναι

```
[2, 3, 4]
[2, 3, 4]
```

Αυτό σημαίνει πως όποιες αλλαγές έγιναν στον πίνακα μέσα στην increment ισχύουν και έξω από αυτήν. Η εξήγηση έχει ως εξής: Η μεταβλητή s είναι μία αναφορά που περιέχει τη διεύθυνση του πίνακα. Η αναφορά αυτή αντιγράφεται στην παράμετρο store της increment. Εφόσον η s είναι αντίγραφο της store, και οι δύο δείχνουν στην ίδια θέση μνήμης. Άρα η increment επενεργεί στον ίδιο πίνακα στον οποίο αναφέρεται η s. Όποια μεταβολή γίνει στα στοιχεία του πίνακα μέσω μιας αναφοράς σε αυτόν, ισχύει όταν τον προσπελάσουμε μέσω οποιασδήποτε άλλης αναφοράς.
Προσέξτε όμως μια άλλη επίπτωση. Μελετήστε τον κώδικα 7.8. Ποια είναι η έξοδός του κατά την άποψή σας;

```
static void assign(int[] store) { //Κώδικας 7.8
    int[] s = {4, 5, 6};
    store = s;
    System.out.println(Arrays.toString(store));
}

public static void main(String[] args) { //Κώδικας 7.8
    int[] table = {1, 2, 3};
    assign(table);
    System.out.println(Arrays.toString(table));
}
```

Κώδικας 7.8 *Η μεταβλητή αναφοράς ως παράμετρος τιμής*

Η έξοδος του κώδικα 7.8 έχει ως εξής:

```
[4, 5, 6]
[1, 2, 3]
```

Η έξοδος αυτή μας δείχνει πως μέσα στην assign, η store μεταβλήθηκε ώστε να δείχνει στον ίδιο πίνακα με την τοπική μεταβλητή s. Ωστόσο, η table συνεχίζει να αναφέρεται στον πίνακα που δημιουργήθηκε στην main. Στην πράξη, ένα αντίγραφο της table πέρασε στην store. Το αντίγραφο, δηλαδή η store, ενημερώθηκε ώστε να δείχνει σε άλλο πίνακα, όμως η table έμεινε ανέπαφη. Το ίδιο ακριβώς αποτέλεσμα θα είχαμε βέβαια αν ονομάζαμε store την table.

Συνοψίζοντας, σε σχέση με τις μεταβλητές αναφοράς ως παραμέτρους, ισχύει το εξής στην Java: Όταν μια μεταβλητή αναφοράς περνάει σαν παράμετρος σε συνάρτηση, η συνάρτηση μπορεί να μεταβάλει το

περιεχόμενο της μνήμης στην οποία αναφέρεται η εν λόγω μεταβλητή, δεν μπορεί όμως να μεταβάλει το περιεχόμενο της ίδιας της μεταβλητής.

7.3.3 Λίστα παραμέτρων μεταβλητού μήκους

Ας υποθέσουμε τώρα πώς αναπτύσσουμε μια εφαρμογή στην οποία απαιτείται συχνά να υπολογίζουμε τον μέσο όρο μιας σειράς πραγματικών τιμών. Το θέμα όμως είναι πως σε ένα σημείο του κώδικα χρειάζεται να υπολογίσουμε τον μέσο όρο τεσσάρων πραγματικών τιμών, σε κάποιο άλλο σημείο χρειάζεται ο μέσος όρος επτά πραγματικών τιμών. Μία λύση θα ήταν να υλοποιήσουμε δύο συναρτήσεις. Η μία με 4 τυπικές παραμέτρους και η άλλη με επτά. Αν αργότερα χρειαστεί να υπολογίσουμε τον μέσο όρο πέντε πραγματικών τιμών, τότε θα πρέπει να προσθέσουμε τον ορισμό κατάλληλης συνάρτησης με πέντε τυπικές παραμέτρους. Η αντιμετώπιση αυτή όμως είναι προβληματική. Σε όλες τις περιπτώσεις, ο αλγόριθμος είναι ο ίδιος και το μόνο που αλλάζει είναι το πλήθος των δεδομένων.

Ευτυχώς για μας, η Java προσφέρει μια αποτελεσματική δυνατότητα, τη λίστα παραμέτρων μεταβλητού μήκους (variable length parameter list).

Ας δούμε στην πράξη πώς υλοποιείται και πώς αξιοποιείται μια συνάρτηση που υπολογίζει τον μέσο όρο μιας σειράς πραγματικών αριθμών.

```
    static double average(double... sequence) {
        double sum = 0;
        for (int i = 0; i < sequence.length; i++) {
            sum += sequence[i];
        }
        return sum / sequence.length;
    }

    public static void main(String[] args) {
        double d1 = 12, d2 = 13.5, d3 = 16.1;
        double av1 = average(d1, 14.5);
        double av2 = average(d1, d2, d3);
        double totalAv = average(av1, av2);
        DecimalFormat f = new DecimalFormat("###.##");
        System.out.println(f.format(av1) + " " + f.format(av2) + " " +
f.format(totalAv));
    }
```

Κώδικας 7.9 *Παράμετρος μεταβλητού μήκους*

Η συνάρτηση average στον κώδικα 7.9 περιέχει την παράμετρο μεταβλητού μήκους sequence. Οι τρεις τελείες που ακολουθούν τον τύπο double δηλώνουν πως πρόκειται για παράμετρο μεταβλητού μήκους. Μόνο μια παράμετρος μεταβλητού μήκους επιτρέπεται σε μία συνάρτηση. Αν μάλιστα η συνάρτηση έχει και άλλες παραμέτρους, τότε η παράμετρος μεταβλητού μήκους πρέπει να βρίσκεται τελευταία στη λίστα παραμέτρων.

Στην main, η average καλείται τρεις φορές. Την πρώτη με δύο πραγματικές παραμέτρους, μία μεταβλητή και μία σταθερά, τη δεύτερη με τρεις πραγματικές παραμέτρους και την τρίτη φορά καλείται πάλι με δύο πραγματικές παραμέτρους.

Μελετώντας την υλοποίηση της average μπορούμε να διαπιστώσουμε πως οι παράμετροι περνάνε στην πράξη σε έναν πίνακα τύπου double. Εκείνο δηλαδή που συμβαίνει είναι πως ο μεταγλωττιστής τοποθετεί τις τιμές των πραγματικών παραμέτρων σε κατάλληλο πίνακα και περνάει μια αναφορά σε αυτόν τον πίνακα στην average. Έτσι μέσα στη συνάρτηση έχουμε πρόσβαση στις τιμές των πραγματικών παραμέτρων μέσω της average την οποία διαχειριζόμαστε κανονικά ως τυπική αναφορά σε μονοδιάστατο πίνακα πραγματικών. Εξαιτίας αυτού του μηχανισμού, μπορούμε να καλέσουμε την average με πραγματική παράμετρο έναν πίνακα πραγματικών όπως δείχνει ο κώδικας που ακολουθεί:

```
double[] d={12,13.5,16.1};
double totalAv=average(d);
```

7.3.4 Οι παράμετροι της main

Μια εφαρμογή Java μπορεί να κληθεί από διάφορα περιβάλλοντα. Στην τυπική περίπτωση, οι εφαρμογές Java σε αυτό το βιβλίο καλούνται μέσα από το NetBeans. Ωστόσο, μια εφαρμογή Java μπορεί να κληθεί μέσα από πολλά και διαφορετικά περιβάλλοντα. Για παράδειγμα, μπορεί να κληθεί από το powershell των Windows ή μέσα από μια άλλη εφαρμογή Java ή μέσα από οποιοδήποτε λειτουργικό στο οποίο τρέχει η JVM.

Επιπλέον, μια εφαρμογή θα πρέπει να έχει τρόπο ακριβώς όπως οι συναρτήσεις να λαμβάνει μια σειρά από παραμέτρους ώστε να διαφοροποιεί τη συμπεριφορά της ανάλογα με τις τιμές των παραμέτρων. Ωστόσο, τα περιβάλλοντα από τα οποία μπορεί να κληθεί μια εφαρμογή Java δεν υποστηρίζουν ακριβώς τους ίδιους τύπους δεδομένων που υποστηρίζει η Java. Προκύπτει επομένως η ανάγκη να υπάρχει μια διεπαφή ικανή να εκκινήσει μια εφαρμογή Java που να είναι ταυτοχρόνως συμβατή με τα διάφορα περιβάλλοντα κλήσης.

Την ανάγκη αυτή καλύπτει η συνάρτηση main. Η main αποτελεί την κύρια είσοδο σε κάθε εφαρμογή Java. Αυτό σημαίνει πως από οποιοδήποτε περιβάλλον και αν καλέσουμε μια εφαρμογή Java, η εικονική μηχανή θα ψάξει να βρει και να καλέσει την main. Στη συνέχεια, η main μπορεί να καλέσει ανάλογα με τον σχεδιασμό της εφαρμογής οποιαδήποτε συνάρτηση είναι στη διάθεση της εφαρμογής. Επομένως, υπάρχει η ανάγκη να παρέχεται ένας τρόπος με τον οποίο τα διάφορα περιβάλλοντα κλήσης να μεταβιβάζουν παραμέτρους στην main. Ένας τύπος δεδομένων διαθέσιμος σε όλα τα περιβάλλοντα είναι η αλφαριθμητική σειρά. Με τη μορφή αλφαριθμητικής σειράς μπορεί να αναπαρασταθούν δεδομένα οποιουδήποτε τύπου. Για παράδειγμα, μια πραγματική τιμή μπορεί να αναπαρασταθεί ως αλφαριθμητική σειρά. Η τιμή 2.53 ας πούμε, μπορεί να αναπαρασταθεί ως “2.53”. Αυτός είναι ο λόγος που η main λαμβάνει ως παράμετρο έναν πίνακα από Strings. Μέσα από την παράμετρο αυτή, τα περιβάλλοντα κλήσης μπορούν να περάσουν τις παραμέτρους που απαιτούνται. Από τη στιγμή που η main θα έχει τις τιμές των παραμέτρων με τη μορφή αλφαριθμητικών τιμών, έχει τη δυνατότητα να τις μετατρέψει στους κατάλληλους τύπους και να εργαστεί με αυτές χωρίς πρόβλημα.

Ας πάρουμε όμως τα πράγματα από την αρχή. Ας θυμηθούμε τη διεπαφή της main.

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

Καταρχάς, στην τυπική περίπτωση, η main δηλώνεται ως public. Αυτό είναι λογικό, διαφορετικά θα ήταν αδύνατο να κληθεί από τρίτους. Βέβαια ο μεταγλωττιστής δεν θα διαμαρτυρηθεί αν η main δηλωθεί με πρόσβαση πακέτου. Ενδεχομένως μάλιστα σε κάποιες ειδικές περιπτώσεις, όπου η main καλείται μέσα από Java, μια τέτοια δήλωση να είναι χρήσιμη. Ούτε και για τους υπόλοιπους προσδιοριστές προσπέλασης διαμαρτύρεται ο μεταγλωττιστής. Ωστόσο, σε όλες τις περιπτώσεις που η main θα τρέξει απευθείας από την εικονική μηχανή και όχι μέσα από κάποιον κώδικα Java, θα πρέπει να χρησιμοποιηθεί ο προσδιοριστής προσπέλασης public. Αν παρόλα αυτά επιχειρήσουμε να τρέξουμε μια εφαρμογή με μη δημόσια main, η εικονική μηχανή θα αποκριθεί με κατάλληλο μήνυμα στο οποίο θα αναφέρει ότι δεν βρίσκει main.

Επιπλέον, η main είναι δηλωμένη ως static. Πρόκειται για απαραίτητη δήλωση. Στην περίπτωση που θα δηλωθεί ως μη στατική συνάρτηση, η εικονική μηχανή θα αποκριθεί και πάλι πως δεν βρίσκει main. Η main ως κύρια είσοδος στην εφαρμογή πρέπει απαραίτητα να δηλωθεί ως στατική συνάρτηση, καθώς κλήση των μη στατικών συναρτήσεων μπορεί να πραγματοποιηθεί μόνο μέσα από εφαρμογή Java και άρα έπεται της εκκίνησης της εφαρμογής.

Επίσης, η main πρέπει να είναι τύπου void. Για αυτήν την απαίτηση ίσως να μην υπάρχει αναγκαιότητα. Θα μπορούσε να επιστρέφει int αντί για void όπως η main της C. Στην Java όμως έγινε αυτή η σχεδιαστική επιλογή. Η εικονική μηχανή ψάχνει μια main τύπου void.

Ας δούμε όμως μια πρότυπη main με παραμέτρους.

```
public class MainParameters {

    private static void syntax() {
        System.out.println("MainParameters int String");
        System.exit(1);
    }

    public static void main(String[] prms) {
        if (prms.length != 2) {
            syntax();
```

```
        }
        int firstArgument = Integer.parseInt(prms[0]);
        System.out.println("first Argument is " + firstArgument
                + "\nSecond Argument is " + prms[1]);
    }
}
```

Κώδικας 7.10 *Παράδειγμα main με παραμέτρους*

Στον κώδικα 7.10 παρουσιάζουμε την κλάση MainParameters η οποία περιέχει main με παραμέτρους. Βέβαια όλες οι main που είναι κύριες είσοδοι στο πρόγραμμα διαθέτουν τυπική παράμετρο τύπου String[]. Όσες είδαμε όμως μέχρι εδώ δεν χρησιμοποιούν την παράμετρο. Αντίθετα, η MainParameters τη χρησιμοποιεί.

Πιο συγκεκριμένα, η main εδώ χρειάζεται δύο παραμέτρους. Η πρώτη πρέπει να είναι τύπου int και η δεύτερη String. Οι τιμές των δύο παραμέτρων περνάνε στην main ως αλφαριθμητικές σειρές διαμέσου του πίνακα prms. Ας σημειωθεί εδώ πως το αναγνωριστικό prms δεν είναι παρά το αναγνωριστικό μιας παραμέτρου. Μπορεί ο προγραμματιστής που υλοποιεί την main να επιλέξει όποιο αναγνωριστικό κρίνει κατάλληλο. Συνήθως για την παράμετρο της main χρησιμοποιείται το αναγνωριστικό args.

Μέσα στον πίνακα prms λοιπόν αναμένει η main του κώδικα 7.10, δύο παραμέτρους. Η πρώτη που θα τοποθετηθεί στη θέση 0 του πίνακα θα αναπαριστά μια ακέραια τιμή και η δεύτερη που θα τοποθετηθεί στη θέση 1 θα είναι μια αλφαριθμητική σειρά.

Το πρώτο πράγμα που θα πρέπει να ελέγξουμε στην main είναι αν όντως κλήθηκε με σωστές παραμέτρους. Πράγματι, στην πρώτη γραμμή της main ελέγχουμε αν η εφαρμογή κλήθηκε με 2 παραμέτρους. Αν όχι, καλούμε τη συνάρτηση syntax.

Η syntax βγάζει ένα μήνυμα που ενημερώνει τον χρήστη πως η εφαρμογή μας καλείται με δύο παραμέτρους, μία τύπου int και μια τύπου String. Στη συνέχεια καλεί την System.exit με παράμετρο 1. Η System.exit προκαλεί άμεσο τερματισμό της εικονικής μηχανής και άρα και της εφαρμογής που την καλεί. Επομένως αν κληθεί η Syntax βγαίνει ένα μήνυμα στον χρήστη που υποδεικνύει τον ορθό τρόπο κλήσης και η εφαρμογή τερματίζεται καθώς χωρίς τις κατάλληλες παραμέτρους αδυνατεί να προχωρήσει.

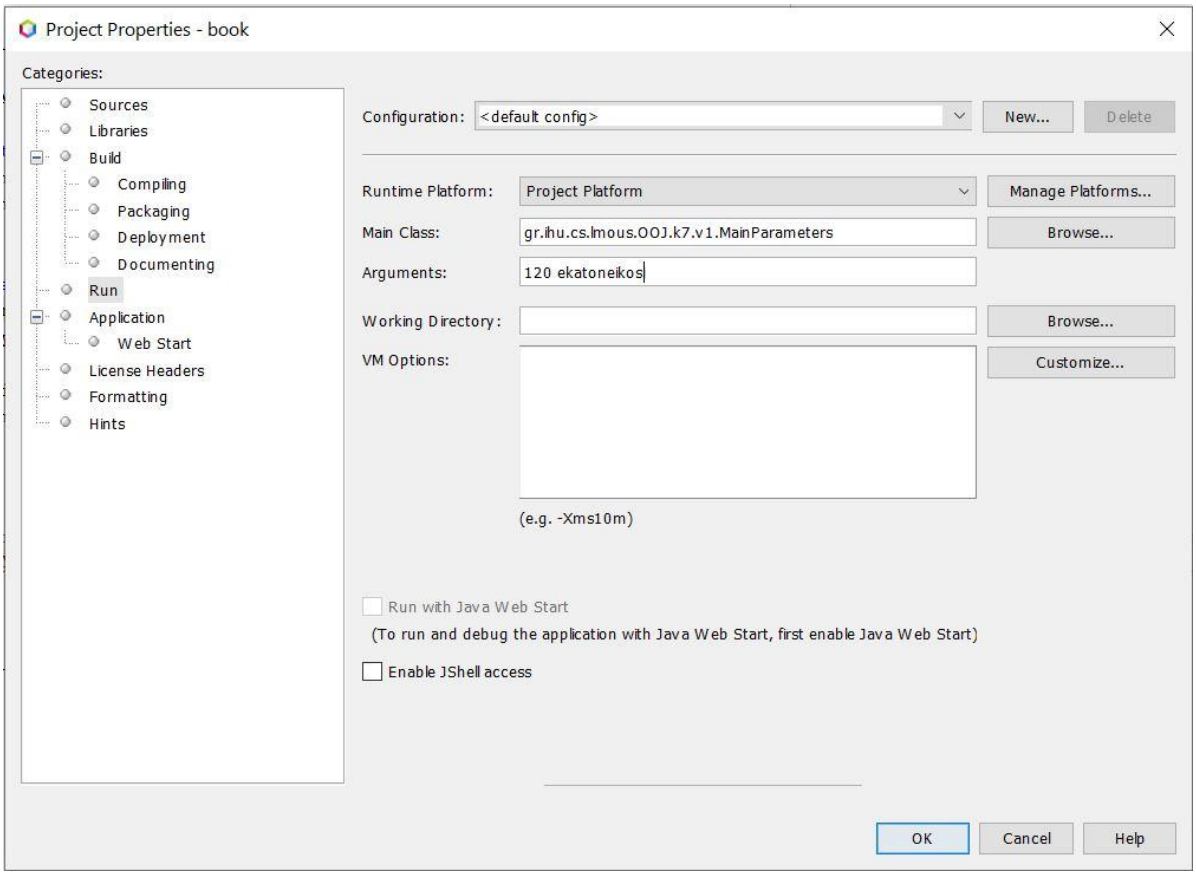
Αν δεν κληθεί η syntax, τότε η εφαρμογή επιχειρεί να μετατρέψει την πρώτη παράμετρο, δηλαδή αυτή που βρίσκεται στη θέση μηδέν του prms σε int. Έτσι όπως είναι ο κώδικας 7.10, αν η πρώτη παράμετρος δεν είναι μετατρέψιμη σε int, το πρόγραμμα πέφτει παράγοντας κατάλληλο μήνυμα. Αν η μετατροπή των παραμέτρων γίνει με επιτυχία, τότε η main έχει τις τιμές των παραμέτρων σε τοπικές μεταβλητές και μπορεί να προχωρήσει απρόσκοπτα στην ενδεδειγμένη επεξεργασία. Στον κώδικα 7.10 η πρώτη παράμετρος μετατρέπεται σε int με τη βοήθεια της Integer.parseInt και η τιμή της αποθηκεύεται στην τοπική μεταβλητή firstArgument. Η δεύτερη παράμετρος είναι τύπου String και δεν απαιτείται καμιά μετατροπή. Μπορεί να χρησιμοποιηθεί η prms[1].

Να επανέλθουμε λίγο στην System.exit και στην τιμή 1 που της περάσαμε ως πραγματική παράμετρο. Όπως έχουμε εξηγήσει, οι συναρτήσεις διαθέτουν μια τιμή επιστροφής που σκοπό έχει να ενημερώνει τον κώδικα που κάλεσε τη συνάρτηση με τα αποτελέσματα της κλήσης. Η main όμως είναι εξ ορισμού void. Η παράμετρος της System.exit αποτελεί έναν τρόπο για να ενημερώνουμε αυτόν που καλεί την εφαρμογή για το αποτέλεσμά της. Σε περίπτωση που η εφαρμογή έτρεξε ομαλά, η τιμή της παραμέτρου πρέπει να είναι μηδέν. Οποιαδήποτε άλλη τιμή μπορεί να χρησιμοποιηθεί για να επικοινωνήσει μη ομαλό τερματισμό της εφαρμογής. Από εκεί και μετά, οι πιθανές τιμές είναι θέμα σύμβασης μεταξύ της εφαρμογής και του περιβάλλοντος που την καλεί. Στα Windows για παράδειγμα, η τιμή της παραμέτρου της System.exit ενημερώνει τη μεταβλητή συστήματος errorlevel. Όλα τα scripts των Windows που καλούν κάποια εφαρμογή Java, μπορούν να ελέγχουν την errorlevel και να λαμβάνουν πληροφορίες σχετικά με τον τερματισμό της εφαρμογής.

Ας δούμε όμως δύο παραδείγματα κλήσης της MainParameters. Πιο συγκεκριμένα, θα δούμε πώς καλείται η MainParameters από το NetBeans και από το powershell των Windows.

7.3.4.1 Κλήση από το NetBeans

Στο τμήμα Projects του NetBeans, μεταβείτε στο project στο οποίο έχετε ορίσει την MainParameters και πατήστε δεξί κλικ. Θα ανοίξει ένα πτυσσόμενο μενού. Επιλέξτε Properties. Θα ανοίξει η φόρμα Project Properties.



Εικόνα 7.1 *Καθορισμός παραμέτρων της main στο NetBeans*

Στην αριστερή πλευρά της φόρμας Project Properties είναι τοποθετημένη μία λίστα κάτω από την επικεφαλίδα Categories. Στη λίστα Categories, επιλέξτε Run όπως φαίνεται στην εικόνα 7.1. Στη δεξιά πλευρά της φόρμας βρείτε τα πεδία Main Class και Arguments και ενημερώστε τα κατάλληλα. Στο πεδίο Main Class εισάγετε την κλάση MainParameters με τον προσδιοριστή πακέτου όπως φαίνεται στην εικόνα 7.1. Ο προσδιοριστής πακέτου έχει το όνομα του πακέτου στο οποίο ανήκει η κλάση. Μπορείτε να χρησιμοποιήσετε το button Browse για διευκόλυνσή σας. Στο πεδίο Arguments, τοποθετήστε τις παραμέτρους που απαιτεί η εφαρμογή διαχωρισμένες με ένα διάστημα. Πατήστε το button OK ώστε να αποθηκευτούν οι νέες ιδιότητες.

Μεταβείτε στο project που έχετε ορίσει την κλάση, πατήστε δεξί κλικ και από το πτυσσόμενο μενού, επιλέξτε Run. Στο παράθυρο Output του NetBeans θα δείτε να εμφανίζεται το αποτέλεσμα κλήσης της MainParameters.

7.3.4.2 Κλήση από το powershell

Μεταβείτε στο Project Properties. Στο παράθυρο Categories, επιλέξτε Sources. Επιλέξτε τη διαδρομή στην οποία είναι αποθηκευμένο το project σας από το πεδίο Project Folder. Χρησιμοποιήστε Ctrl+c για να αντιγράψετε τη διαδρομή του project στο πρόχειρο των Windows (clipboard).

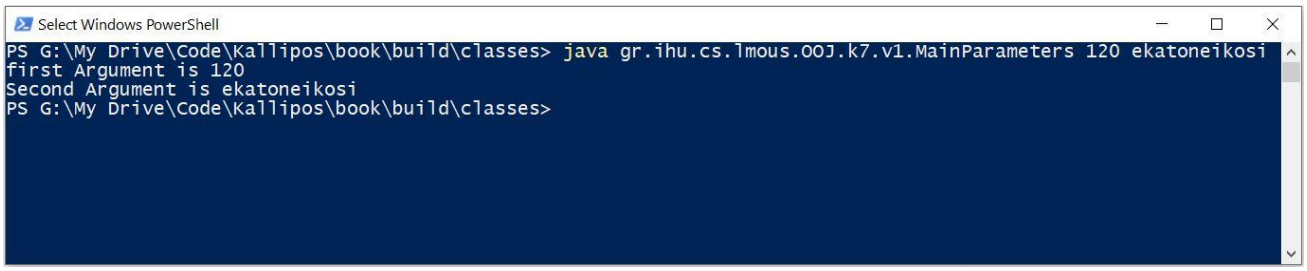
Ανοίξτε τον file explorer των Windows. Στη γραμμή εντολών δώστε Ctrl+V και Enter. Θα μεταβείτε έτσι στον φάκελο του project. Εκεί θα δείτε μια σειρά από φακέλους. Ο φάκελος build περιέχει φάκελο classes. Στον φάκελο classes είναι αποθηκευμένος ο bytecode της εφαρμογής σας οργανωμένος σε υποφακέλους ανάλογα με το πακέτο στο οποίο ανήκει η κάθε κλάση.

Μεταβείτε στον φάκελο classes. Ενώ βρίσκεστε στον φάκελο classes, πηγαίνετε στο μενού file του file explorer και κάντε κλικ στο Open Windows PowerShell. Θα ανοίξει το PowerShell με φάκελο εργασίας τον φάκελο classes.

Εκεί πληκτρολογήστε

```
Java packagePath.MainParameters first-argument second-argument
```

Στη θέση packagePath βάλτε το όνομα του πακέτου στο οποίο ανήκει η MainParameters. Η εικόνα 7.2 παρουσιάζει μια κλήση της MainParameter όταν αυτή έχει δηλωθεί να ανήκει στο πακέτο gr.ihu.cs.lmous.OOJ.k7.v1.



Εικόνα 7.2 *Κλήση εφαρμογής Java από το PowerShell των Windows.*

Όπως φαίνεται στην εικόνα 7.2, σε αυτό το παράδειγμα, η MainParameters καλείται με πρώτη παράμετρο το 112 και δεύτερη παράμετρο τη σειρά ekatondwdeka. Η λέξη Java στην αρχή της κλήσης ενεργοποιεί την JVM η οποία στην ουσία τρέχει την κλάση MainParameters.

7.4 Στατικές μεταβλητές

Η Java μας δίνει τη δυνατότητα να ορίσουμε στατικές μεταβλητές. Οι στατικές μεταβλητές ορίζονται στο επίπεδο της κλάσης και προσπελαύνονται μέσα από οποιαδήποτε συνάρτηση της κλάσης. Δεν υπάρχει δυνατότητα ορισμού στατικών μεταβλητών που είναι τοπικές μεταβλητές συναρτήσεων. Η πλήρης έννοια των στατικών μεταβλητών μπορεί να αναπτυχθεί μόνο στο πλαίσιο του Αντικειμενοστρεφούς μοντέλου όπου θα συζητηθούν και οι μη στατικές μεταβλητές και θα γίνει διαφοροποίηση μεταξύ στατικών και μη στατικών μεταβλητών.

Για την ώρα θα δούμε τη χρησιμότητα των στατικών μεταβλητών μέσα από ένα παράδειγμα αξιοποίησής των. Ας υποθέσουμε πως στην εφαρμογή που αναπτύσσουμε χρειαζόμαστε σε μορφή αλφαριθμητικής σειράς το άθροισμα ή και το γινόμενο πραγματικών αριθμών αποθηκευμένων σε κατάλληλο πίνακα. Σε αυτήν την περίπτωση είναι λογικό να υλοποιήσουμε δύο συναρτήσεις, μία που υπολογίζει και επιστρέφει το άθροισμα και μια για το γινόμενο. Θέλουμε όμως οι δύο συναρτήσεις να διαμορφώνουν τα αποτελέσματά τους με κοινό μοτίβο. Σκοπός μας είναι να εμφανίσουμε άθροισμα και γινόμενο και δεν επιθυμούμε να εμφανίζονται με διαφορετικό πλήθος δεκαδικών. Επομένως χρειαζόμαστε έναν τρόπο διαμόρφωσης του αποτελέσματος κοινό για τις δύο συναρτήσεις. Εδώ μας διευκολύνουν οι στατικές μεταβλητές της κλάσης:

```
import java.text.DecimalFormat;

public class StaticVars {

    public static String format; // "###.##"

    public static String sum(double[] in) {
        double sum = 0;
        for (double current : in) {
            sum += current;
        }
        DecimalFormat f = new DecimalFormat(format);
        return f.format(sum);
    }

    public static String product(double[] in) {
        double product = 1;
```

```
        for (double current : in) {
            product *= current;
        }
        DecimalFormat f = new DecimalFormat(format);
        return f.format(product);
    }

    public static void main(String[] args) {
        format = "###.##";
        double[] d = {5.6, 7.95, 3.23, 45, 96};
        System.out.println(sum(d));
        System.out.println(product(d));
    }
}
```

Κώδικας 7.11 *Χρήση στατικής μεταβλητής*

Στον κώδικα 7.11 δίνεται ο ορισμός της κλάσης StaticVars. Η κλάση περιλαμβάνει ορισμό των συναρτήσεων sum και product. Και οι δύο λαμβάνουν ως παράμετρο έναν πίνακα πραγματικών. Η sum υπολογίζει το άθροισμα των τιμών της παραμέτρου της και η product το γινόμενο. Και οι δύο υπολογίζουν το αποτέλεσμά τους και το επιστρέφουν ως String. Στην κλάση ορίζεται και η στατική μεταβλητή format. Σκοπός της είναι να ελέγχει τη διαμόρφωση κατά την εμφάνιση των πραγματικών μεταβλητών. Ως στατική μεταβλητή της κλάσης, η format μπορεί να προσπελαστεί από όλες τις συναρτήσεις της κλάσης. Στην main, πριν καλέσουμε τις μεθόδους, φροντίζουμε να ενημερώσουμε κατάλληλα την format. Στη συνέχεια, οι δύο μέθοδοι διαβάζουν την format και επιστρέφουν το αποτέλεσμά τους διαμορφωμένο σύμφωνα με το μοτίβο που αποθηκεύσαμε στην format.

Κατά μία έννοια, μια στατική μεταβλητή αποτελεί μία κοινή μνήμη για όλη την κλάση.

Ας υποθέσουμε τώρα πως χρειαζόμαστε την ίδια λειτουργικότητα, δηλαδή να εμφανίζουμε το άθροισμα και το γινόμενο των στοιχείων πινάκων πραγματικών και σε μία άλλη εφαρμογή. Έστω όμως πως στην άλλη εφαρμογή, χρειαζόμαστε διαφορετική διαμόρφωση, π.χ. θέλουμε τρία δεκαδικά ψηφία αντί για δύο. Με δεδομένο ότι οι συναρτήσεις sum και product αλλά και η σειρά format είναι δημόσια μέλη της κλάσης, η λύση είναι απλή και παρατίθεται στον κώδικα 7.12.

```
public class AnotherApp {

    public static void main(String[] args) {
        StaticVars.format = "###.###";
        double[] d = {5.675, 7.915, 3.213, 45, 0.96};
        System.out.println(StaticVars.sum(d));
        System.out.println(StaticVars.product(d));
    }
}
```

Κώδικας 7.12 *Χρήση στατικής μεταβλητής από άλλη κλάση*

Στον κώδικα 7.12, η main της κλάσης AnotherApp δίνει τιμή στη στατική μεταβλητή format της StaticVars. Στη συνέχεια καλεί τις sum και product της StaticVars προκειμένου να εμφανίσει το άθροισμα και το γινόμενο των στοιχείων του πίνακα d.

Όπως θα αντιληφθήκατε ήδη, προκειμένου να προσπελάσουμε μια στατική συνάρτηση ή μια στατική μεταβλητή από κλάση διαφορετική από αυτήν που ορίζονται πρέπει να προηγηθεί του ονόματός τους, το όνομα της κλάσης στην οποία ορίζονται. Σε αυτήν τη χρήση, το όνομα της κλάσης θεωρείται ένα προσδιοριστικό ονόματος (name specifier).

Αν έχουμε σύγκρουση μεταξύ αναγνωριστικού στατικής μεταβλητής μιας κλάσης με τοπική μεταβλητή μέσα σε συνάρτηση της ίδιας κλάσης, τότε θα πρέπει να χρησιμοποιήσουμε τον προσδιοριστή ονόματος κατά την αναφορά στη στατική μεταβλητή.

7.5 Απροσδόκητα λάθη

Ας υποθέσουμε πως θέλουμε να αναπτύξουμε μια συνάρτηση που υπολογίζει το παραγοντικό μιας ακέραιας τιμής. Το πρώτο βήμα βέβαια θα είναι να βεβαιωθούμε πως γνωρίζουμε τον σωστό ορισμό του παραγοντικού.

Για τη διευκόλυνση της συζήτησής μας παραθέτουμε τον ορισμό.

Το παραγοντικό ενός μη αρνητικού ακέραιου n, συμβολίζεται με n! και είναι ίσο με 1 αν n=0 ή n=1, ενώ αν n>1, είναι το γινόμενο όλων των θετικών ακέραιων που είναι μικρότεροι ή ίσοι με το n.

Με δεδομένο τον ορισμό, η ανάπτυξη της συνάρτησης είναι σχετικά απλή. Ο κώδικας 7.13 μας δίνει τη συνάρτηση του παραγοντικού.

```
public class UnexpectedErrors {

    static long factorial(long n) { //Κώδικας 7.13
        if (n == 0 || n == 1) {
            return 1;
        }
        long rVal = 1;
        for (long i = 2; i <= n; i++) {
            rVal *= i;
        }
        return rVal;
    }
}
```

Κώδικας 7.13 *Υπολογισμός παραγοντικού*

Η συνάρτηση factorial είναι τύπου long καθώς το παραγοντικό μεγαλώνει γρήγορα σε σχέση με το όρισμά του. Για παράδειγμα, 3! = 6, 6! = 720 και 9! = 362880. Στην αρχή της συνάρτηση factorial, ελέγχεται το n καθώς αν είναι 0 ή 1 δεν απαιτούνται περαιτέρω υπολογισμοί μια και για αυτές τις τιμές του n, το n! είναι ίσο με 1. Στη συνέχεια αρχικοποιείται η rVal στην τιμή 1 και πολλαπλασιάζεται διαδοχικά με όλους τους ακέραιους που είναι μεγαλύτεροι ή ίσοι του 2 και μικρότεροι ή ίσοι του n. Με την έξοδο από τον βρόχο for, η rVal αναπαριστά την απαιτούμενη τιμή την οποία η συνάρτηση επιστρέφει.

Αν τρέξετε την factorial, θα διαπιστώσετε ότι επιστρέφει τη σωστή τιμή για όλες τις μη αρνητικές τιμές της παραμέτρου n.

Τι θα συμβεί όμως αν η factorial κληθεί με πραγματική παράμετρο έναν αρνητικό ακέραιο; Καταρχάς, δεν θα επιστρέψει από την αρχική if. Στη συνέχεια θα αρχικοποιήσει την rVal στο 1. Στη συνέχεια δεν θα μπει στον βρόχο for, καθώς το θετικό i δεν είναι μικρότερο ή ίσο από οποιονδήποτε αρνητικό αριθμό και τέλος θα επιστρέψει το rVal του οποίου η τιμή παραμένει 1. Με λίγα λόγια, η factorial του κώδικα 7.13 επιστρέφει την τιμή 1 ως αποτέλεσμα υπολογισμού του παραγοντικού οποιουδήποτε αρνητικού ακέραιου.

Αυτό όμως είναι λάθος, μιας και δεν ορίζεται παραγοντικό για αρνητικούς ακέραιους. Πρόκειται μάλιστα για σοβαρό λάθος με απρόβλεπτες συνέπειες κατά την εκτέλεση της εφαρμογής που το περιέχει.

Μπορεί να αναρωτιέστε γιατί να κληθεί η factorial με αρνητικό ακέραιο. Ενδεχομένως, η factorial να καλείται με όρισμα κάποια μεταβλητή η οποία να προκύπτει από κάποιους υπολογισμούς. Ενδεχομένως, το λάθος να ξεκινά από εκεί. Γενικά, στις ρεαλιστικές εφαρμογές όπου εμπλέκονται πάρα πολλές μεταβλητές και πάρα πολλοί και σύνθετοι υπολογισμοί, τέτοιου είδους λάθη είναι σχεδόν αδύνατο να αποκλειστούν. Εκείνο όμως που μπορεί να γίνει είναι να εντοπιστούν εγκαίρως και να διορθωθούν. Σε αυτό μας βοηθάνε οι εξαιρέσεις (Exceptions) της Java.

Η διαχείριση των απροσδόκητων λαθών είναι μεγάλο κομμάτι κάθε γλώσσας προγραμματισμού και της Java. Αντίστοιχα, πολλές είναι και οι κλάσεις που αφορούν τη διαχείριση των λαθών. Υπάρχουν διαχειρίσιμα και μη διαχειρίσιμα λάθη. Λεπτομέρειες για τη διαχείριση των λαθών συζητάμε στην ενότητα 16. Για την ώρα μας αρκεί μια προσέγγιση που αν και απλή μας βοηθάει να προστατευτούμε από τα απροσδόκητα λάθη στον κώδικά μας. Η προσέγγιση λοιπόν που προτείνουμε εδώ βασίζεται στην παραγωγή μιας εξαίρεσης τύπου RuntimeException. Με άλλα λόγια, στην αρχή της συνάρτησης factorial θα ελέγξουμε το όρισμα και αν βρεθεί αρνητικό θα παράγουμε μια εξαίρεση τύπου RuntimeException. Η παραγωγή της εξαίρεσης θα προκαλέσει τερματισμό του προγράμματος με εμφάνιση κατάλληλου μηνύματος. Έτσι θα βοηθηθούμε να εντοπίσουμε το πρόβλημα και να το διορθώσουμε.

Καλό είναι αυτός που καλεί τη συνάρτησή μας να γνωρίζει πως η συνάρτηση ενδέχεται να παράγει εξαίρεση. Έτσι, η Java υποστηρίζει τη δήλωση των εξαιρέσεων που ενδέχεται να παράγει μια συνάρτηση

στην επικεφαλίδα της. Ο κώδικας 7.14 παρουσιάζει την factorial με ενσωματωμένες τις παρατηρήσεις που κάναμε σε αυτήν την ενότητα.

```
public class UnexpectedErrors {

    static long factorial(long n) throws RuntimeException { //Κώδικας 7.14
        if (n < 0) {
            throw new RuntimeException();
        }
        if (n == 0 || n == 1) {
            return 1;
        }
        long rVal = 1;
        for (long i = 2; i <= n; i++) {
            rVal *= i;
        }
        return rVal;
    }
}
```

Κώδικας 7.14 *Η factorial με έλεγχο της παραμέτρου*

Όπως φαίνεται στον κώδικα 7.14, η παραγωγή της εξαίρεσης γίνεται με τη λέξη-κλειδί throw. Με τη βοήθεια της λέξης-κλειδί throws δηλώνεται προαιρετικά στην επικεφαλίδα της συνάρτησης το γεγονός ότι η συνάρτηση ενδέχεται να παράγει εξαίρεση τύπου RuntimeException.

Τρέξτε τη νέα έκδοση της factorial με αρνητικό όρισμα και μελετήστε το αποτέλεσμα.

7.6 Χρήσιμες συναρτήσεις

Σε αυτήν την ενότητα παρουσιάζουμε συναρτήσεις ενσωματωμένες στο βασικό πλαίσιο της Java που συχνά μας είναι χρήσιμες. Όπως θα έχετε ήδη αντιληφθεί, κάθε συνάρτηση στην Java ανήκει και σε μία κλάση. Επιπρόσθετα λοιπόν κάποιων συναρτήσεων της βιβλιοθήκης της Java που έχουμε ήδη δει, στην ενότητα αυτή θα δούμε κάποιες συναρτήσεις που ανήκουν στις κλάσεις Math και String.

7.6.1 Η κλάση Math

Η κλάση Math περιλαμβάνει μια σειρά από στατικές συναρτήσεις και στατικές σταθερές μεταβλητές που μας επιτρέπουν να κάνουμε βασικούς μαθηματικούς υπολογισμούς. Παρουσιάζουμε εδώ μέρος των συναρτήσεων και σταθερών μεταβλητών της Math. Αυτές που θεωρούμε πως είναι πιο αναγκαίες στα πλαίσια αυτού του εγχειριδίου. Για κάθε μια συνάρτηση παρουσιάζουμε πρώτα τη διεπαφή της ακολουθούμενη από μια μικρή περιγραφή και στη συνέχεια επιδεικνύουμε τον τρόπο κλήσης της.
Η σταθερά E αναπαριστά το e, τη βάση των φυσικών λογάριθμων. Η διεπαφή της έχει ως εξής:

```
public static final double E
```

Η σταθερά PI αναπαριστά το π, την αναλογία της περιφέρειας ενός κύκλου προς τη διάμετρο του κύκλου. Η διεπαφή της είναι:

```
public static final double PI
```

Η συνάρτηση toRadians μετατρέπει τις μοίρες σε ακτίνια.

```
public static double toRadians(double angdeg)
```

Η παράμετρος angdeg αναπαριστά τις μοίρες μιας γωνίας. Η συνάρτηση επιστρέφει τα ακτίνια της ίδιας γωνίας.

Η συνάρτηση toDegrees μετατρέπει τα ακτίνια σε μοίρες.

```
public static double toDegrees(double angrad)
```

Η παράμετρος angrad αναπαριστά τα ακτίνια μιας γωνίας. Η συνάρτηση επιστρέφει τις μοίρες της ίδιας γωνίας.

Η συνάρτηση sin επιστρέφει το ημίτονο μιας γωνίας.

```
public static double sin(double a)
```

Η παράμετρος a αναπαριστά τα ακτίνια μιας γωνίας. Αν επομένως θέλετε να υπολογίσετε το ημίτονο μιας γωνίας 30 μοιρών, δεν έχετε παρά να καλέσετε

```
Math.sin(Math.toRadians(30))
```

Η συνάρτηση cos επιστρέφει το συνημίτονο μιας γωνίας.

```
public static double cos(double a)
```

Η παράμετρος a αναπαριστά τα ακτίνια μιας γωνίας. Αν επομένως θέλετε να υπολογίσετε το συνημίτονο των 30 μοιρών καλέστε

```
Math.cos(Math.toRadians(30))
```

Η συνάρτηση tan επιστρέφει την εφαπτομένη γωνίας.

```
public static double tan(double a)
```

Η παράμετρος a αναπαριστά τα ακτίνια μιας γωνίας. Αν επομένως θέλετε να υπολογίσετε την εφαπτομένη των 30 μοιρών καλέστε

```
Math.tan(Math.toRadians(30))
```

Η συνάρτηση log επιστρέφει τον φυσικό λογάριθμο της παραμέτρου της.

```
public static double log(double a)
```

Για να υπολογίσετε τον φυσικό λογάριθμο του 90, καλέστε

```
Math.log(90)
```

Η συνάρτηση log10 επιστρέφει τον λογάριθμο με βάση το 10.

```
public static double log10(double a)
```

Για να υπολογίσετε τον λογάριθμο με βάση 10 του 90, καλέστε

```
Math.log10(90)
```

Η συνάρτηση pow επιστρέφει τη δύναμη ενός αριθμού υψωμένου σε έναν εκθέτη.

```
public static double pow(double a,  double b)
```

Η παράμετρος a αναπαριστά τη βάση και η b τον εκθέτη. Για να υπολογίσετε την παράσταση 5.24 καλέστε

Math.pow(5.2, 4)

Η συνάρτηση sqrt επιστρέφει την τετραγωνική ρίζα του ορίσματός της.

```
public static double sqrt(double a)
```

Για να υπολογίσετε την τετραγωνική ρίζα του 81, καλέστε

```
Math.sqrt(81)
```

Η συνάρτηση ceil επιστρέφει τη μικρότερη ακέραια τιμή που είναι μεγαλύτερη ή ίση με την παράμετρό της.

```
public static double ceil(double a)
```

Η κλήση ceil(9.1) επιστρέφει 10d, ενώ η κλήση ceil(9.0) επιστρέφει 9d.

Η συνάρτηση floor επιστρέφει τη μεγαλύτερη ακέραια τιμή που είναι μικρότερη ή ίση με την παράμετρό της.

```
public static double floor(double a)
```

Η κλήση floor(9.1) επιστρέφει 9d όπως και η κλήση floor(9.0).

Η συνάρτηση rint επιστρέφει την ακέραια τιμή που είναι πλησιέστερη στην τιμή της παραμέτρου της.

```
public static double rint(double a)
```

Η κλήση rint(9.4) επιστρέφει 9d και η κλήση rint(9.6) επιστρέφει 10d. Αν η τιμή της παραμέτρου βρίσκεται ακριβώς στη μέση μεταξύ δύο ακέραιων τιμών, τότε η rint επιστρέφει τον ακέραιο που είναι άρτιος. Επομένως, η κλήση rint(9.5) θα επιστρέψει 10d όπως ακριβώς και η κλήση rint(10.5).

Η συνάρτηση round επιστρέφει την ακέραια τιμή που είναι πλησιέστερη στην τιμή της παραμέτρου της. Η συνάρτηση είναι υπερφορτωμένη. Αν κληθεί με όρισμα double, επιστρέφει long και αν κληθεί με όρισμα float, επιστρέφει int. Αν η τιμή της παραμέτρου βρίσκεται ακριβώς στη μέση μεταξύ δύο ακέραιων τιμών, τότε η round επιστρέφει τον μεγαλύτερο από τους πλησιέστερους ακέραιους.

```
public static long round(double a)
public static int round(float a)
```

Η κλήση Math.round(9.4) επιστρέφει 9 και η κλήση Math.round(9.5) επιστρέφει 10.

Η συνάρτηση max επιστρέφει το μεγαλύτερο από τα ορίσματά της. Η συνάρτηση είναι υπερφορτωμένη και υποστηρίζει όλους τους αριθμητικούς τύπους. Οι διαθέσιμες διεπαφές έχουν ως εξής:

```
public static double max(double a, double b)
public static float max(float a, float b)
public static int max(int a, int b)
public static long max(long a, long b)
```

Αν καλέσετε την max με ορίσματα διαφορετικού τύπου, τότε θα κληθεί η έκδοση που είναι συμβατή και με τους δύο τύπους. Για παράδειγμα, αν την καλέσετε με double και long, τότε θα κληθεί η max(double, double). Έτσι η έκφραση

```
double m=Math.max(1d, 2L);
```

```
είναι σωστή. Ενώ η έκφραση
```

```
long m=Math.max(1d, 2L);
```

είναι λάθος.

Ακριβώς ανάλογη με την max είναι η min.

Η συνάρτηση abs επιστρέφει την απόλυτη τιμή της παραμέτρου της. Η abs είναι επίσης υπερφορτωμένη για όλους τους αριθμητικούς τύπους. Οι διαθέσιμες διεπαφές έχουν ως εξής:

```
public static double abs(double a)
public static float abs(float a)
public static long abs(long a)
public static int abs(int a)
```

7.6.2 Οι κλάσεις των θεμελιωδών τύπων

Έχουμε ήδη εξηγήσει πως για κάθε θεμελιώδη τύπο, η Java παρέχει μια αντίστοιχη κλάση. Κάθε μια τέτοια κλάση παρέχει σταθερές και συναρτήσεις που είναι χρήσιμες κατά την εκτέλεση εργασιών με θεμελιώδεις τύπους. Σε αυτήν την ενότητα συγκεντρώνουμε τις σημαντικότερες στατικές συναρτήσεις αυτών των κλάσεων.

7.6.2.1 Η κλάση Character

Η συνάρτηση isDigit επιστρέφει true αν η παράμετρος της αναπαριστά ψηφίο σύμφωνα με την κωδικοποίηση Unicode. Επομένως, εκτός από τα ψηφία που χρησιμοποιούνται στη Δυτική γραφή, υποστηρίζονται και τα ψηφία από διάφορους πολιτισμούς. Για παράδειγμα, οι χαρακτήρες '٢', '٣' και '٤' αποτελούν ψηφία στο αραβικό σύστημα αρίθμησης. Η διεπαφή της συνάρτησης είναι

```
public static boolean isDigit(char ch)
```

Η συνάρτηση isLetter επιστρέφει true αν η παράμετρος της αναπαριστά κάποιο γράμμα σύμφωνα με την κωδικοποίηση Unicode. Προφανώς, η συνάρτηση δεν είναι αληθής μόνο για τους λατινικούς χαρακτήρες. Για παράδειγμα, οι χαρακτήρες '暘', '暲' και '晴' αναπαριστούν γράμματα στη μανδαρίνικη γραφή. Η διεπαφή της συνάρτησης είναι

public static boolean isLetter(char ch)

Η συνάρτηση isLowerCase επιστρέφει true αν η παράμετρος της αναπαριστά πεζό γράμμα.

public static boolean isLowerCase(char ch)

Η συνάρτηση isSpaceChar επιστρέφει true αν η παράμετρος της αναπαριστά χαρακτήρα που είναι διαχωριστικό διαστήματος, γραμμής ή παραγράφου. Υπάρχουν συνολικά 19 τέτοιοι χαρακτήρες στο Unicode με κωδικούς 32, 160, 5760, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8232, 8233, 8239, 8287 και 12288. Η διεπαφή της συνάρτησης είναι

public static boolean isSpaceChar(char ch)

Η συνάρτηση isUpperCase επιστρέφει true αν η παράμετρος της αναπαριστά κεφαλαίο γράμμα. Η διεπαφή της συνάρτησης είναι

public static boolean isUpperCase(char ch)

Η συνάρτηση toString επιστρέφει την παράμετρό της ως String μήκους 1. Η διεπαφή της συνάρτησης είναι

public static String toString(char c)

7.6.2.2 Οι κλάσεις Integer, Double και Boolean

Καταρχάς να επισημάνουμε πως ό,τι αναφέρουμε σε αυτήν την ενότητα για την κλάση Integer ισχύει κατ' αναλογία και για τις κλάσεις Byte, Short και Long. Επίσης ό,τι αναφέρεται για την κλάση Double ισχύει κατ' αναλογία και για την κλάση Float.

Οι συναρτήσεις

```
public static int parseInt(String s) throws
NumberFormatException
public static double parseDouble(String s) throws
NumberFormatException
public static boolean parseBoolean(String s)
```

ανήκουν στις κλάσεις Integer, Double και Boolean, αντίστοιχα και μετατρέπουν το όρισμά τους σε int, double και boolean, αντίστοιχα.

Προσέξτε πως οι parseInt και parseDouble ενδέχεται να παράξουν λάθος τύπου NumberFormatException. Πρόκειται για λάθος στενά συγγενικό με το RuntimeException που είδαμε στην ενότητα 7.5. Αντίθετα, η parseBoolean δεν παράγει κανενός τύπου λάθος. Η διαφορά αυτή εξηγείται ως εξής: Η parseInt για παράδειγμα, για να λειτουργήσει σωστά θα πρέπει το όρισμά της να είναι μετατρέψιμο σε ακέραιο. Για παράδειγμα, η σειρά "*12.2" δεν μπορεί να μετατραπεί σε ακέραιο. Αν επομένως κληθεί η parseInt με αυτήν τη σειρά θα παραγάγει λάθος.

Αντίθετα η parseBoolean επιστρέφει true μόνο αν το όρισμά της είναι η σειρά "true" ανεξαρτήτως πεζών ή κεφαλαίων και false σε όλες τις άλλες περιπτώσεις.

Η συνάρτηση

```
public static int compare(double d1, double d2)
```

της κλάσης Double, συγκρίνει δύο πραγματικές μεταβλητές ή σταθερές και επιστρέφει 0 αν το d1 είναι αριθμητικά ίσο με το d2, τιμή μικρότερη του μηδενός αν το d1 είναι αριθμητικά μικρότερο από το d2, και τιμή μεγαλύτερη από το 0 αν το d1 είναι αριθμητικά μεγαλύτερο από το d2. Η συνάρτηση θεωρεί πως η τιμή Double.NaN είναι μεγαλύτερη από όλες τις άλλες τιμές τύπου Double και πως το θετικό 0d είναι μεγαλύτερο από το αρνητικό -0d. Διαφέρει η compare σε αυτό το σημείο σε σχέση με τους τελεστές ισότητας και ανισότητας <, <=, ==, >=, >. Για παράδειγμα ο κώδικας

```
System.out.println(Double.NaN>Double.MAX_VALUE);
System.out.println(Double.compare(Double.NaN,Double.MAX_VALUE)
);
System.out.println(+0d>-0d);
System.out.println(Double.compare(+0d, -0d));
```

εμφανίζει

```
false
1
false
1
```

Επομένως ο τελεστής μεγαλύτερο από δεν βγάζει την Double.NaN μεγαλύτερη από την Double.MAX_VALUE ενώ η compare την βγάζει. Παρομοίως, ο τελεστής > δεν διαφοροποιεί μεταξύ των 2 μηδέν ενώ η compare διαφοροποιεί.

Οι συναρτήσεις

```
public static boolean isNaN(double v)
```

```
public static boolean isInfinite(double v)
```

επιστρέφουν true αν το όρισμά τους είναι Double.NaN ή Double.POSITIVE_INFINITY και Double.NEGATIVE_INFINITY, αντίστοιχα.

Τέλος, όλες οι κλάσεις των θεμελιωδών τύπων διαθέτουν τη στατική συνάρτηση toString παρόμοια με την toString της Character, ενότητα 7.6.2.1, που επιστρέφει την τιμή της παραμέτρου της με τη μορφή αλφαριθμητικής σειράς.

7.6.3 Η κλάση String

Η συνάρτηση

```
public static String copyValueOf(char[] data)
```

επιστρέφει ένα String που συντίθεται από τους χαρακτήρες του πίνακα data.
Η συνάρτηση

```
public static String copyValueOf(char[] data, int offset, int
count)
```

επιστρέφει ένα String που συντίθεται από count στον αριθμό χαρακτήρες του πίνακα data αρχίζοντας από τη θέση offset.

Οι συναρτήσεις valueOf επιστρέφουν την String αναπαράσταση του ορίσματός τους. Η valueOf είναι υπερφορτωμένη για όλους τους θεμελιώδεις τύπους. Ενδεικτικά παραθέτουμε τη διεπαφή της συνάρτησης για τον τύπο double

```
public static String valueOf(double d)
```

7.7 Προσδιοριστές προσπέλασης

Έχουμε συζητήσει αρκετά για τα στατικά μέλη της κλάσης. Εκείνο όμως που δεν έχουμε ξεκαθαρίσει είναι με ποια κριτήρια θα καθορίζουμε τον τύπο της πρόσβασης σε αυτά.

Δυστυχώς για αυτήν την απόφαση δεν υπάρχουν κατηγορηματικοί κανόνες και τα πάντα εξαρτώνται από τον στόχο που εξυπηρετεί η κάθε συνάρτηση. Ωστόσο, μια συζήτηση επάνω σε αυτό το θέμα μπορεί να μας βοηθήσει να λαμβάνουμε σωστότερες σχετικές αποφάσεις και να προστατεύουμε τους χρήστες των συναρτήσεών μας.

Η συζήτηση εδώ περιορίζεται στον έλεγχο της πρόσβασης σε στατικές συναρτήσεις και μεταβλητές.

Η Java υποστηρίζει 4 προσδιοριστές προσπέλασης, την ιδιωτική (private), τη δημόσια (public), την προσπέλαση πακέτου (package access) ή εξορισμού (default access) και την προστατευμένη (protected).

Για την προστατευμένη πρόσβαση δεν μπορούμε να πούμε και πολλά. Πρόκειται για τύπο πρόσβασης που σχετίζεται πολύ στενά με την κληρονομικότητα και παρουσιάζεται αναλυτικά στην ενότητα 13.1.2. Εκείνο όμως που μας ενδιαφέρει σχετικά με τα στατικά μέλη είναι πως παρότι ο μεταγλωττιστής δεν διαμαρτύρεται, σε γενικές γραμμές, η προστατευμένη πρόσβαση δεν ταιριάζει στα στατικά μέλη καθώς αυτά δεν υπακούουν στους κανόνες της κληρονομικότητας.

Πρόσβαση πακέτου δηλώνουμε όταν ένα μέλος της κλάσης θέλουμε να μπορεί να χρησιμοποιηθεί από όλες τις κλάσεις του ίδιου πακέτου αλλά όχι από κλάσεις που ανήκουν σε άλλα πακέτα. Θα μπορούσαμε να πούμε πως ό,τι είναι η ιδιωτική πρόσβαση σε επίπεδο κλάσης είναι και η πρόσβαση πακέτου σε επίπεδο πακέτου. Συχνά μάλιστα αναφέρεται ως ιδιωτική πρόσβαση πακέτου (package-private).

Ας δούμε τώρα ένα παράδειγμα με δύο συναρτήσεις όπου η μια είναι καλύτερο να δηλωθεί ως ιδιωτική και η άλλη ως δημόσια. Έστω λοιπόν πως θέλουμε να υλοποιήσουμε μια βιβλιοθήκη μαθηματικών συναρτήσεων μέσα στην οποία περιλαμβάνεται μια συνάρτηση που υπολογίζει έναν πραγματικό αριθμό υψωμένο σε ακέραια δύναμη. Για αυτό το παράδειγμα, εκπαιδευτικού χαρακτήρα, υποθέτουμε επίσης πως η συνάρτηση pow της Math δεν είναι διαθέσιμη,

Σε γενικές γραμμές είναι εύκολο να υλοποιήσουμε μια τέτοια συνάρτηση. Ας δούμε μια πρώτη προσέγγιση:

```
static double powerPos(double b, int e) { //Κώδικας 7.15
    if (e == 0) {
        return 1;
    }
    double rVal = 1;
    for (int i = 0; i < e; i++) {
        rVal *= b;
    }
    return rVal;
}
```

Κώδικας 7.15 *Συνάρτηση που υπολογίζει την τιμή πραγματικού υψωμένου σε μη αρνητικό ακέραιο εκθέτη*

Στον κώδικα 7.15, η συνάρτηση powerPos επιστρέφει την τιμή της πρώτης παραμέτρου της υψωμένη στην τιμή της δεύτερης παραμέτρου της. Ωστόσο, η συνάρτηση επιστρέφει σωστά αποτελέσματα μόνο για μη αρνητικούς εκθέτες. Αν την καλέσετε για οποιονδήποτε αρνητικό εκθέτη, θα σας επιστρέψει λανθασμένα την τιμή 1. Αυτό μπορείτε να αντιληφθείτε εύκολα διαβάζοντας τον κώδικα 7.15. Εφόσον η if δεν θα οδηγήσει σε επιστροφή της συνάρτησης, η συνάρτηση θα προχωρήσει αρχικοποιώντας τη μεταβλητή rVal στο 1. Στη συνέχεια, η ροή δεν μπαίνει καθόλου στον βρόχο for μια και για την αρχική τιμή 0 του i δεν αληθεύει η i<e. Επομένως, η συνάρτηση καταλήγει να επιστρέφει 1.

Το γεγονός ότι η συνάρτηση μπορεί να παράγει λάθος για αρνητικούς εκθέτες την καθιστά εντελώς ακατάλληλη για δημόσια χρήση. Ωστόσο, αυτός που υλοποιεί την κλάση των μαθηματικών συναρτήσεων και γνωρίζει το συγκεκριμένο πρόβλημα θα μπορούσε να αξιοποιήσει τη συνάρτηση ώστε να διευκολυνθεί στην ανάπτυξη μιας πιο ολοκληρωμένης συνάρτησης υπολογισμού της δύναμης. Να τονίσουμε πάντως πως μια τέτοια συνάρτηση όπως η powerPos θα πρέπει να χρησιμοποιείται μόνο με πολλή προσοχή και πολύ περιορισμένα. Ας δούμε όμως πώς μπορεί να βοηθήσει η powerPos στην ανάπτυξη ορθής συνάρτησης υπολογισμού δύναμης.

```
static double power(double b, int e) { // Κώδικας 7.16
    if (e >= 0) {
        return powerPos(b, e);
    }
    return 1 / powerPos(b, -e);
}
```

Κώδικας 7.16 *Συνάρτηση που υπολογίζει την τιμή πραγματικού υψωμένου σε ακέραιο εκθέτη*

Όπως φαίνεται στον κώδικα 7.16, η συνάρτηση power ελέγχει την παράμετρο e που αντιπροσωπεύει τον εκθέτη. Αν η τιμή της είναι μη αρνητική, καλεί την powerPos και επιστρέφει σωστά τη δύναμη μη αρνητικού εκθέτη. Αν η τιμή της είναι αρνητική, δεν μπαίνει μέσα στην if, οπότε προχωρά και καλεί την powerPos με παράμετρο -e και επιστρέφει και πάλι σωστό αποτέλεσμα για αρνητικό εκθέτη.

Αυτό είναι ένα χαρακτηριστικό παράδειγμα όπου η μεν powerPos επιβάλλεται να δηλωθεί ως ιδιωτική ώστε να προστατεύεται ο χρήστης της βιβλιοθήκης μας από πιθανά λανθασμένη κλήση, ενώ η power μπορεί να δηλωθεί ως δημόσια ώστε να μπορεί ο χρήστης να υπολογίσει τις δυνάμεις που επιθυμεί.

7.8 Πακέτα

Ένα πακέτο συντίθεται ως ένα σύνολο κλάσεων ή και άλλων τύπων[6]. Επομένως, ένα πακέτο είναι ταυτόχρονα ένα σύνολο αρχείων πηγαίου κώδικα. Ο σκοπός του είναι να παρέχει έλεγχο πρόσβασης στα μέλη του και έναν μοναδικό ονοματοχώρο (namespace). Τα μέλη ενός πακέτου καθορίζονται από τη δήλωση που γίνεται υποχρεωτικά στην πρώτη γραμμή του αντίστοιχου αρχείου. Για τον έλεγχο πρόσβασης έχουμε ήδη μιλήσει στην ενότητα 7.7. Ας δούμε μερικές άλλες χρήσιμες λεπτομέρειες.

[6] Διεπαφών (interface), Απαριθμήσιμων τύπων (enumerated types) και τύπων σχολίων (annotation types).

Η ανάπτυξη κώδικα στις μέρες μας είναι μια κατεξοχήν συνεργατική εργασία. Έτσι, οι προγραμματιστές που εργάζονται σε μία εταιρεία, συχνά χρησιμοποιούν κώδικες που έχουν αναπτύξει συνάδελφοί τους αλλά και κώδικες που έχουν αναπτυχθεί εκτός της εταιρείας. Προκύπτει εδώ ένα πρόβλημα. Τι θα συμβεί αν στον κώδικά μου έχω ορίσει μια συνάρτηση με ίδια ταυτότητα με μια συνάρτηση που έχει ορισθεί σε κώδικα τρίτου που όμως τον χρησιμοποιώ και εγώ; Για παράδειγμα, στον κώδικά μου χρησιμοποιώ τη συνάρτηση public static int f(int). Παράλληλα, σε έναν άλλο κώδικα που χρειάζομαι είναι ορισμένη συνάρτηση με την ίδια ταυτότητα σε ομώνυμη κλάση. Όταν καλώ π.χ. f(5), ποια από τις δύο f θα τρέξει; Τη λύση εδώ δίνει το πακέτο. Αν η δική μου f έχει ορισθεί στο πακέτο a και η άλλη f στο πακέτο b, τότε μπορώ να διαφοροποιήσω καλώντας a.ClassName.f(5) ή b.ClassName.f(5). Πώς όμως θα αποφύγουμε να έχουν και τα δύο πακέτα την ίδια ονομασία; Αυτό επιτυγχάνεται αποκλειστικά μέσω συμβάσεων για την ονοματολογία των πακέτων. Πιο συγκεκριμένα, ισχύουν οι ακόλουθες συμβάσεις:

1. Στην ονομασία ενός πακέτου θα πρέπει να χρησιμοποιούνται μόνο πεζοί χαρακτήρες ώστε να αποφεύγεται σύγκρουση με ονόματα κλάσεων ή άλλων τύπων.
2. Στην ονομασία ενός πακέτου πρέπει να συμμετέχει ένας τομέας διαδικτύου (internet domain). Οι επαγγελματίες συνηθίζουν να χρησιμοποιούν τη διεύθυνση email αντεστραμμένη όπως κάνουμε στους κώδικες αυτού του βιβλίου, π.χ. gr.ihu.cs.lmous.OOJ.k2.v2. Η οργάνωση των πακέτων και κλάσεων μέσα σε αυτά ώστε να μην προκύπτει σύγκρουση σε αναγνωριστικά είναι υπόθεση που εύκολα ελέγχεται σε συγκεκριμένο τομέα. Το θέμα είναι πως ένας τομέας δικτύου είναι υποχρεωτικά μοναδικός, οπότε έτσι εξασφαλίζεται και η μοναδικότητα στην ονομασία του πακέτου.
3. Σε πολλές περιπτώσεις, η επαναληπτική χρήση του ονόματος ενός πακέτου μέσα στον κώδικα είναι κουραστική. Έτσι η Java μας δίνει τη δυνατότητα να εισάγουμε διάφορα μέλη ενός πακέτου στον κώδικά μας και να τα χρησιμοποιούμε χωρίς την αναφορά πακέτου. Η εισαγωγή επιτυγχάνεται με τη χρήση της δεσμευμένης λέξης import.

Έτσι για παράδειγμα η πρόταση

```
import a.ClassName;
```

μας επιτρέπει να χρησιμοποιήσουμε τη συνάρτηση f της ClassName χωρίς την αναφορά πακέτου, δηλαδή ως ClassName.f(5).

Αν πάλι θέλουμε να εισάγουμε όλους τους τύπους από το πακέτο a, τότε θα πρέπει να χρησιμοποιήσουμε την import ως ακολούθως:

```
import a.*;
```

Προσοχή όμως, ο αστερίσκος δεν έχει το σύνηθες νόημα. Αν θέλουμε να εισάγουμε όλους τους τύπους του πακέτου a που το όνομά τους αρχίζει από B, η πρόταση

```
import a.B*;
```

δεν ισχύει και θα προκαλέσει λάθος μεταγλώττισης.

Επίσης, η πρόταση

```
import a.*;
```

εισάγει όλους τους τύπους του πακέτου a αλλά δεν εισάγει τίποτε από τα πακέτα a.b, a.c. Αν θέλουμε να εισάγουμε όλους τους τύπους του πακέτου a και όλους τους τύπους του πακέτου a.b, τότε πρέπει να χρησιμοποιήσουμε δύο import όπως φαίνεται παρακάτω.

```
import a.*;
import a.b.*;
```

7.9 Λυμένες ασκήσεις

Σε αυτήν την ενότητα παρουσιάζουμε μια σειρά από παραδείγματα στατικών συναρτήσεων ώστε να βοηθήσουμε τον αναγνώστη να εξοικειωθεί τόσο με την υλοποίηση όσο και με τη χρήση τους.

7.9.1 Μέγιστη τιμή πίνακα

Να υλοποιηθεί συνάρτηση που λαμβάνει ως παράμετρο έναν μονοδιάστατο πίνακα ακέραιων και επιστρέφει τη θέση της μέγιστης τιμής του.

Λύση
Καταρχάς θα πρέπει να αποσαφηνίσουμε τη διεπαφή της συνάρτησης. Σύμφωνα με την εκφώνηση, η συνάρτηση λαμβάνει έναν μονοδιάστατο πίνακα ακέραιων. Επομένως, η λίστα παραμέτρων είναι σαφής. Επίσης, σύμφωνα με την εκφώνηση, η συνάρτηση επιστρέφει το μέγιστο στοιχείο του πίνακα. Επομένως, επιστρέφει έναν ακέραιο. Ο κώδικας 7.17 παρουσιάζει τη συνάρτηση.

```
public static int max(int[] tbl) { //Κώδικας 7.17
    int max = tbl[0];
    int idx = 0;
    for (int i = 1; i < tbl.length; i++) {
        if (tbl[i] > max) {
            idx = i;
            max = tbl[i];
        }
    }
    return idx;
}
```

Κώδικας 7.17 *Συνάρτηση εύρεσης της μέγιστης τιμής σε πίνακα ακέραιων*

Για την υλοποίηση της συνάρτησης χρησιμοποιούμε δύο βοηθητικές μεταβλητές, την max και την idx. Η max χρησιμεύει για την αποθήκευση της τιμής του μεγαλύτερου στοιχείου και η idx για την αποθήκευση της θέσης του μεγαλύτερου στοιχείου. Η idx αρχικοποιείται ώστε να αναφέρεται στην αρχή του πίνακα και η max στην τιμή του στοιχείου στην θέση 0.

Στη συνέχεια, σε μια επαναληπτική διαδικασία ελέγχονται ένα προς ένα τα υπόλοιπα στοιχεία του πίνακα. Κάθε φορά που το υπό έλεγχο στοιχείο είναι μεγαλύτερο από την τιμή της max ενημερώνεται τόσο η max όσο και η idx. Στο τέλος του βρόχου for, η max είναι ενημερωμένη με την τιμή του μεγαλύτερου στοιχείου του πίνακα και η idx με τη θέση του.

Προσέξτε πως η for ξεκινάει από 1 και όχι από 0 όπως συνήθως. Αν ξεκινούσε από 0, στο πρώτο επαναληπτικό βήμα θα εξέταζε αν το στοιχείο στη θέση 0 είναι μεγαλύτερο από την τιμή της max. Η max όμως έχει αρχικοποιηθεί στην τιμή του στοιχείου στη θέση 0. Επομένως, πρόκειται για περιττή σύγκριση.

Επίσης, ενδιαφέρον έχει πως η συνάρτηση επιστρέφει τη θέση της μεγαλύτερης τιμής και όχι την ίδια την τιμή. Πράγματι, είναι πιο χρήσιμη η θέση παρά η τιμή καθώς αν έχω τη θέση, εύκολα μπορώ να προσπελάσω την τιμή.

Τέλος, ας σημειωθεί πως αν στον πίνακα υπάρχουν περισσότερα από ένα μέγιστα στοιχεία, η υλοποίηση του κώδικα 7.17 επιστρέφει τη θέση του πρώτου από τα μέγιστα.

7.9.2 Εξίσωση β' βαθμού

Να υλοποιηθεί συνάρτηση που επιστρέφει τη λύση εξίσωσης β' βαθμού.

Λύση
Καταρχάς θα πρέπει να δούμε πώς θα μοντελοποιήσουμε το πρόβλημα, δηλαδή στην προκειμένη περίπτωση να αποφασίσουμε ποια θα είναι η διεπαφή της συνάρτησης. Γνωρίζουμε από τα μαθηματικά πώς η εξίσωση β' βαθμού έχει τη μορφή

$$ax^2 + bx + c = 0, με\ a, b, c\ \in R\ και\ a \neq 0$$

Επομένως, η εξίσωση ορίζεται από τους συντελεστές της. Αυτούς πρέπει να περάσουμε παραμετρικά στη συνάρτηση. Τι όμως θα επιστρέφει η συνάρτηση; Όπως γνωρίζουμε, οι λύσεις εξαρτώνται από την τιμή της διακρίνουσας. Αν η διακρίνουσα, Δ, είναι μεγαλύτερη από το 0, τότε έχουμε δύο διακριτές πραγματικές λύσεις, αν η διακρίνουσα είναι ίση με το 0, τότε έχουμε μια κοινή λύση και αν η διακρίνουσα είναι μικρότερη από το 0, η εξίσωση δεν έχει λύση.

Μία αντιμετώπιση είναι να επιστρέφουμε έναν πίνακα πραγματικών μεγέθους 2. Στην περίπτωση που $\Delta > 0$, τότε στην πρώτη θέση του πίνακα θα έχουμε τη μία ρίζα και στη δεύτερη την άλλη, αν $\Delta = 0$ και στις δύο θέσεις θα έχουμε την κοινή ρίζα, τέλος αν $\Delta < 0$, τότε και στις δύο θέσεις θα τοποθετήσουμε την τιμή Double.NaN.

Βέβαια όλα αυτά για να τα κάνουμε θα πρέπει να υπολογίσουμε τη διακρίνουσα. Παρόλο που ο χρήστης της συνάρτησης θα πληροφορηθεί για το πρόσημο της διακρίνουσας, δεν θα γνωρίζει την ακριβή τιμή της. Μία εναλλακτική προσέγγιση είναι η συνάρτησή μας να επιστρέφει πίνακα μήκους 3, όπου στη θέση 0 να τοποθετεί την τιμή της διακρίνουσας. Στον κώδικα 7.18 δίνεται αυτή η τελευταία εκδοχή της συνάρτησης.

```
static double[] triwnymo(double a, double b, double c) { // Κώδικας 7.18
    if (a == 0) {
        throw new RuntimeException();
    }
    double[] rVal = new double[3];
    double d = b * b - 4 * a * c;
    rVal[0] = d;
    if (d < 0) {
        rVal[1] = rVal[2] = Double.NaN;
        return rVal;
    }
    rVal[1] = (-b + Math.sqrt(d)) / (2 * a);
    rVal[2] = (-b - Math.sqrt(d)) / (2 * a);
    return rVal;
}
```

Κώδικας 7.18 *Συνάρτηση λύσης εξίσωσης β' βαθμού*

Σε αυτήν την περίπτωση η εξίσωση μπορεί να κληθεί όπως δείχνει ο κώδικας 7.19.

```
static void klisiTriwnymo() { // Κώδικας 7.19
    double[] solution = triwnymo(2, 5, -1);
    if (solution[0] < 0) {
        System.out.println("Η Εξίσωση σεν έχει λύση στο R");
    } else {
        System.out.println("χ1=" + solution[1] + " x2=" + solution[2]);
    }
}
```

Κώδικας 7.19 *Κλήση της συνάρτησης triwnymo*

Μία εναλλακτική κλήση δίνεται στον κώδικα 7.20

```
static void klisiTriwnymo2() { // Κώδικας 7.20
    double[] solution = triwnymo(2, 5, -1);
    if (Double.isNaN(solution[1])) {
        System.out.println("Η Εξίσωση σεν έχει λύση στο R");
    } else {
        System.out.println("x1=" + solution[1] + " x2=" + solution[2]);
    }
}
```

Κώδικας 7.20 *Εναλλακτική κλήση της συνάρτησης triwnymo*

Προσοχή όμως δεν είναι σωστή η κλήση

```
if (solution[1]==Double.NaN)….
```

καθώς εξορισμού η έκφραση Double.NaN==Double.NaN είναι ψευδής.
Σε μια εναλλακτική προσέγγιση μπορούμε να αναπαραστήσουμε το τριώνυμο ως έναν πίνακα πραγματικών μήκους 3. Η συνάρτηση μπορεί να υπερφορτωθεί εύκολα ώστε να υποστηρίζεται και αυτή η αναπαράσταση του τριωνύμου, όπως δείχνει ο κώδικας 7.21.

```
static double[] triwnymo(double[] syn) { // Κώδικας 7.21
    return triwnymo(syn[0], syn[1], syn[2]);
}
```

Κώδικας 7.21 *Υπερφόρτωση της συνάρτησης triwnymo*

Σε αυτήν την περίπτωση, γίνεται η αναπαράσταση εξισώσεων β' βαθμού ευκολότερη, όπως δείχνει ο κώδικας 7.22.

```
double[] ex1 = {2, 5, -1}, ex2 = {4, 4, -9};
double[] solution1 = triwnymo(ex1);
double[] solution2 = triwnymo(ex2);
```

Κώδικας 7.22 *Αναπαράσταση εξισώσεων β βαθμού ως double[]*

7.9.3 Fibonacci

Να αναπτυχθεί συνάρτηση που επιστρέφει τον ν-οστό όρο της ακολουθίας Fibonacci.

Λύση
Καταρχάς θα πρέπει να δούμε ποιος ακριβώς είναι ο ορισμός της ακολουθίας Fibonacci. Στο διαδίκτυο μπορούμε εύκολα να τον βρούμε. Αν συμβολίσουμε τον ν-οστό με Fn, τότε ο ορισμός της ακολουθίας έχει ως εξής:

$$n = 0 \rightarrow F_n = 0$$
$$n = 1 \rightarrow F_n = 1$$
$$n > 1 \rightarrow F_n = F_{n-1} + F_{n-2}$$

Επομένως, η συνάρτησή μας θα πρέπει να λαμβάνει μία ακέραια παράμετρο, μπορεί όμως να επιστρέφει ένα long, δεδομένου πως η τιμή του ν-οστού όρου είναι σημαντικά μεγαλύτερη από το n. Ο κώδικας 7.23 δίνει μια υλοποίηση της συνάρτησης.

```
static long fibonacci(int fibIdx) { //Κώδικας 7.23
    if (fibIdx < 0) {
        throw new RuntimeException();
    } else if (fibIdx <= 1) {
        return fibIdx;
    }
    int n = 2;
    long current = 1, previous = 1, beforePrevious;
    while (++n <= fibIdx) {
        beforePrevious = previous;
        previous = current;
        current = beforePrevious + previous;
    }
    return current;
}
```

Κώδικας 7.23 *Υλοποίηση της συνάρτησης Fibonacci*

Καθώς δεν υπάρχει αρνητικός όρος της ακολουθίας, αν η συνάρτηση κληθεί με παράμετρο μικρότερη του μηδενός, σημαίνει πως κάποιο λάθος έχει συμβεί. Διαφορετικά αν η παράμετρος fibIdx είναι μικρότερη ή ίση με το 1, δηλαδή αν είναι 0 ή 1, βάσει ορισμού, επιστρέφεται η ίδια η τιμή της παραμέτρου. Αν παρόλα αυτά δεν επιστρέψει η συνάρτηση σημαίνει ότι κλήθηκε με παράμετρο μεγαλύτερη του ένα, οπότε η τιμή του ν-οστού όρου πρέπει να υπολογιστεί ως το άθροισμα των δύο προηγούμενων όρων. Η μεταβλητή n αναπαριστά την τάξη του όρου και ξεκινά από το 2, οι δε μεταβλητές current, previous και beforePrevious αναπαριστούν τις τιμές του ν-οστού όρου, του προηγούμενού του και του όρου πριν τον προηγούμενο του ν-οστού. Ο βρόχος while επισκέπτεται έναν-έναν τους όρους της ακολουθίας έως ότου φτάσει στον όρο που εκφράζει η παράμετρος fibIdx. Σε κάθε βήμα αυτής της επαναληπτικής διαδικασίας ενημερώνονται κατάλληλα οι προηγούμενοι όροι. Με την έξοδο από την while, η current είναι ενημερωμένη με την τιμή του όρου στη θέση fibIdx.

7.9.4 Λεκτικό αριθμού

Να υλοποιηθεί συνάρτηση static String lektiko(int num) που δέχεται μια ακέραια παράμετρο από 0 έως 999 και επιστρέφει τη διατύπωσή της στα Ελληνικά.

Λύση

Σύμφωνα με την εκφώνηση και με τη διεπαφή που δίνεται μια κλήση όπως η lektiko(125) θα πρέπει να επιστρέψει την αλφαριθμητική σειρά “εκατόν είκοσι πέντε”. Εδώ έχουμε δύο θέματα να μας απασχολήσουν. Πρώτον, πόσα και ποια λεκτικά μας είναι απαραίτητα για να σχηματίσουμε την απαιτούμενη διατύπωση όλων των αριθμών από 0 έως 999 και δεύτερον πώς θα εξάγουμε πόσες εκατοντάδες έχουμε, πόσες δεκάδες και πόσες μονάδες. Προσέξτε πως αν διαιρέσετε έναν ακέραιο με το 100 (ακέραια διαίρεση), τότε λαμβάνεται τον αριθμό εκατοντάδων που περιέχει. Αν στη συνέχεια βγάλετε τις εκατοντάδες, μένει το υπόλοιπο του ακέραιου. Για παράδειγμα: Έστω num=475, h ο αριθμός των εκατοντάδων και r το υπόλοιπο, τότε h=num/100=4, r=num-h*100=75. Στη συνέχεια διαιρούμε το υπόλοιπο με το 10 για να λάβουμε τον αριθμό των δεκάδων. Τέλος, οι μονάδες είναι το υπόλοιπο μετά την αφαίρεση των δεκάδων. Ο κώδικας 7.24 επιστρέφει το λεκτικό ενός ακέραιου από το 0 έως το 999.

```
static String lektiko(int num) { //Κώδικας 7.24
    if (num < 0 || num > 999) {
        throw new RuntimeException();
    }
    if (num == 0) {
        return "μηδέν";
    }
    int ekatontades = num / 100;
    num -= ekatontades * 100;
    int dekades = num / 10;
    num -= dekades * 10;
    String hundreds = "", tens = "", units = "";
    switch (ekatontades) {
        case 0:
            break;
        case 1:
            hundreds = "εκατόν";
            break;
        case 2:
            hundreds = "διακόσια";
            break;
        case 3:
            hundreds = "τριακόσια";
            break;
        case 4:
            hundreds = "τετρακόσια";
            break;
        case 5:
            hundreds = "πεντακόσια";
```

```
            break;
        case 6:
            hundreds = "εξακόσια";
            break;
        case 7:
            hundreds = "επτακόσια";
            break;
        case 8:
            hundreds = "οκτακοσια";
            break;
        case 9:
            hundreds = "εννιακόσια";
            break;
    }
    switch (dekades) {
        case 0:
            break;
        case 1:
            tens = "δέκα";
            break;
        case 2:
            tens = "είκοσι";
            break;
        case 3:
            tens = "τριάντα";
            break;
        case 4:
            tens = "σαράντα";
            break;
        case 5:
            tens = "πενήντα";
            break;
        case 6:
            tens = "εξήντα";
            break;
        case 7:
            tens = "εβδομήντα";
            break;
        case 8:
            tens = "ογδόντα";
            break;
        case 9:
            tens = "ενενήντα";
            break;
    }
    switch (num) {
        case 0:
            break;
        case 1:
            if (tens.equals("δέκα")) {
                tens = "έντεκα";
            } else {
                units = "ένα";
            }
            break;
        case 2:
            if (tens.equals("δέκα")) {
                tens = "δώδεκα";
            } else {
                units = "δύο";
            }
```

```
                break;
            case 3:
                units = "τρία";
                break;
            case 4:
                units = "τέσσερα";
                break;
            case 5:
                units = "πέντε";
                break;
            case 6:
                units = "έξη";
                break;
            case 7:
                units = "επτά";
                break;
            case 8:
                units = "οκτώ";
                break;
            case 9:
                units = "εννέα";
                break;
        }
        String rVal = hundreds + (tens != "" ? " " : "") + tens + (units != "" ? 
" " : "") + units;
        //return rVal;
        return rVal.trim();
    }
```

Κώδικας 7.24 *Μετατροπή ακέραιου σε λεκτικό*

Προσέξτε την έκφραση tens!=””?” ”:””. Σημαίνει πως αν η μεταβλητή tens έχει πάρει τιμή διάφορη από ““, τότε επιστρέφουμε ένα διάστημα που θα διαχωρίσει τις εκατοντάδες από τις δεκάδες. Αν όμως η tens παραμένει ““, τότε σημαίνει πως δεν έχουμε δεκάδες, επομένως δεν υπάρχει λόγος να προσθέσουμε το αντίστοιχο διάστημα. Ίδια λογική ισχύει και για την units!=””?” ”:””.

7.9.5 Κλάση MyArrays

Να υλοποιηθεί η κλάση MyArrays που περιλαμβάνει τις ακόλουθες στατικές συναρτήσεις χωρίς να χρησιμοποιηθεί η κλάση Arrays.

```
public static int seqSearch(int[] t, int sE)
```

Πραγματοποιεί σειριακή αναζήτηση στον t για την τιμή sE. Επιστρέφει τη θέση στον t της sE εφόσον αυτή βρεθεί, διαφορετικά -1.

```
public static int seqSearch(String[] t, String sE)
```

Πραγματοποιεί σειριακή αναζήτηση στον t για την τιμή sE. Επιστρέφει τη θέση στον t της sE εφόσον αυτή βρεθεί, διαφορετικά -1.

```
public static int cntOccurances(int[] t, int sE)
```

Μετράει και επιστρέφει το πλήθος τιμών sE στον t.

```
public static String toString(int[] t)
```

Επιστρέφει μια String αναπαράσταση του t.

```
public static String toString(int[][] t)
```

Επιστρέφει μια String αναπαράσταση του t.

```
public static int[] reverse(int[] t)
```

Επιστρέφει έναν πίνακα ακέραιων που περιέχει τα στοιχεία του t με αντεστραμμένη διάταξη.

```
public static int sum(int[] t)
```

Επιστρέφει το άθροισμα των στοιχείων του t.

```
public static int max(int[] t)
```

Επιστρέφει τη θέση του μέγιστου στοιχείου του t.

```
public static boolean equals(int[] tA, int[] tB)
```

Επιστρέφει true αν κάθε στοιχείο του tA είναι και στοιχείο του tB και κάθε στοιχείο του tB είναι και στοιχείο του tA.

```
public static boolean equals(String[] tA, String[] tB)
```

Επιστρέφει true αν κάθε στοιχείο του tA είναι και στοιχείο του tB και κάθε στοιχείο του tB είναι και στοιχείο του tA.

```
public static int binarySearch(int[] t, int sE)
```

Πραγματοποιεί δυαδική αναζήτηση στον t για την τιμή sE. Αν η τιμή βρεθεί, επιστρέφει τη θέση της στον t, διαφορετικά επιστρέφει -1.

```
public static void swap(int[] t, int idx1, int idx2)
```

Αντιμεταθέτει το στοιχείο στη θέση idx1 του t με το στοιχείο στη θέση idx2.

```
public static int idxOfMin(int[] array, int startIdx)
```

Επιστρέφει τη θέση του ελάχιστου στοιχείου του t από τη θέση startIdx και μετά.

```
public static void sort(int[] t)
```

Ταξινομεί κατ'αύξουσα αριθμητική σειρά των πίνακα t.

Λύση
Ο κώδικας 7.25 παραθέτει μια λύση της άσκησης και ακολουθούν σχόλια.

```
public class MyArrays {

    public static int seqSearch(int[] t, int sE) {
        for (int i = 0; i < t.length; i++) {
            if (t[i] == sE) {
                return i;
            }
        }
        return -1;
    }
```

```
public static int seqSearch(String[] t, String sE) {
    for (int i = 0; i < t.length; i++) {
        if (t[i].equals(sE)) {
            return i;
        }
    }
    return -1;
}

public static int cntOccurances(int[] t, int sE) {
    int cnt = 0;
    for (int e : t) {
        if (e == sE) {
            cnt++;
        }
    }
    return cnt;
}

public static String toString(int[] t) {
    if (t.length == 0) {
        return "[]";
    }
    String s = "[";
    for (int i = 0; i < t.length; i++) {
        s += t[i];
        if (i < t.length - 1) {
            s += ", ";
        }
    }
    s += "]";
    return s;
}

public static String toString(int[][] t) {
    if (t.length == 0) {
        return "[]";
    }
    String s = "[";
    for (int i = 0; i < t.length; i++) {
        s += toString(t[i]);
        if (i < t.length - 1) {
            s += ",\n";
        }
    }
    s += "]";
    return s;
}

public static int[] reverse(int[] input) {
    int[] rVal = new int[input.length];
    for (int i = 0; i < input.length; i++) {
        rVal[rVal.length - 1 - i] = input[i];
    }
    return rVal;
}

public static int sum(int[] t) {
    int sum = 0;
    for (int i : t) {
```

```
            sum += i;
        }
        return sum;
    }

    public static int max(int[] t) {
        int max = t[0];
        int idx = 0;
        for (int i = 1; i < t.length; i++) {
            if (t[i] > max) {
                idx = i;
                max = t[i];
            }
        }
        return idx;
    }

    public static boolean equals(int[] tA, int[] tB) {
        if (tA.length != tB.length) {
            return false;
        }
        for (int cE : tA) {
            if (seqSearch(tB, cE) == -1) {
                return false;
            }
        }
        return true;
    }

    public static boolean equals(String[] tA, String[] tB) {
        if (tA.length != tB.length) {
            return false;
        }
        for (String cE : tA) {
            if (seqSearch(tB, cE) == -1) {
                return false;
            }
        }
        return true;
    }

    public static int binarySearch(int[] tbl, int schElement) {
        int from = 0, to = tbl.length - 1, mid = (to - from) / 2;
        while (to >= from) {
            if (tbl[mid] == schElement) {
                return mid;
            } else if (schElement < tbl[mid]) {
                to = mid - 1;
            } else {
                from = mid + 1;
            }
            mid = from + (to - from) / 2;
        }
        return -1;
    }

    public static void swap(int[] t, int idx1, int idx2) {
        int i = t[idx1];
        t[idx1] = t[idx2];
        t[idx2] = i;
    }
```

```
    public static int idxOfMin(int[] t, int startIdx) {
        int min = t[startIdx];
        int rVal = startIdx;
        for (int i = startIdx + 1; i < t.length; i++) {
            if (t[i] < min) {
                min = t[i];
                rVal = i;
            }
        }
        return rVal;
    }

    public static void sort(int[] t) {
        for (int i = 0; i < t.length; i++) {
            int mIdx = idxOfMin(t, i);
            if (mIdx != i) {
                swap(t, mIdx, i);
            }
        }
    }

}
```

Κώδικας 7.25 *Η κλάση MyArrays*

Οι δύο πρώτες συναρτήσεις είναι απλές. Η πρώτη κάνει σειριακή αναζήτηση σε πίνακα ακέραιων και η δεύτερη σε πίνακα αλφαριθμητικών σειρών. Η μόνη λεπτομέρεια που πρέπει να προσέξουμε είναι πως για μεν τους ακέραιους, η σύγκριση γίνεται με τον τελεστή ==, για δε τις αλφαριθμητικές σειρές, η σύγκριση γίνεται με τη συνάρτηση equals.

Η cntOccurances επίσης βασίζεται σε σειριακή προσπέλαση όλων των στοιχείων του πίνακα. Κάθε φορά που βρίσκει τιμή ίση με τη δεύτερη παράμετρό της, αυξάνει έναν μετρητή τον οποίον και επιστρέφει.

Η toString για μονοδιάστατο πίνακα ακέραιων, καταρχάς επιστρέφει [] αν η παράμετρός της έχει μήκος μηδέν. Στη συνέχεια, αρχικοποιεί την τοπική μεταβλητή s στη σειρά [. Μετά προσπελαύνει ένα προς ένα τα στοιχεία τα στοιχεία του t και προσθέτει στο s την τιμή του στοιχείου ακολουθούμενη από κόμμα και διάστημα. Εξαιρεί όμως την πρόσθεση κόμματος και διαστήματος στο τελευταίο στοιχείο. Τέλος, κλείνει το s προσθέτοντας το].

Η toString για δισδιάστατο πίνακα έχει ανάλογη λογική με την toString για μονοδιάστατο. Μόνο που εδώ τα στοιχεία είναι πίνακες, οπότε χρησιμοποιεί την toString για να πάρει την αναπαράσταση των μονοδιάστατων πινάκων στοιχείων του t.

Η reverse ορίζει τον πίνακα rVal με μήκος ίσο με το μήκος του πίνακα input. Στη συνέχεια αντιγράφει ένα προς ένα τα στοιχεία του input στον rVal σε αντιδιαμετρικά αντίθετη θέση. Προσέξτε πως όταν ο δείκτης i είναι ίσος με μηδέν, δηλαδή αναφερόμαστε στην πρώτη θέση του input, η έκφραση rVal.length-1-i αναφέρεται στην τελευταία θέση του rVal, όταν ο δείκτης i είναι ίσος με 1, δηλαδή αναφερόμαστε στη δεύτερη θέση του input, η έκφραση rVal.length-1-i αναφέρεται στην προτελευταία θέση του rVal, κοκ.

Η sum εφαρμόζει σειριακή προσπέλαση και προσθέτει κάθε τιμή του t στην τοπική μεταβλητή sum την οποία τελικά επιστρέφει.

Παρόμοια η max εφαρμόζει σειριακή προσπέλαση. Αρχικοποιεί την τοπική μεταβλητή max στην τιμή στη θέση μηδέν του t. Στη συνέχεια, επισκέπτεται τα στοιχεία του t από τη θέση 1 και μετά και κάθε φορά που βρίσκει μεγαλύτερη τιμή ενημερώνει την max και την idx που κρατάει τη θέση της max. Επιστρέφει την idx.

Η equals για πίνακες ακέραιων συγκρίνει καταρχάς τα μήκη των πινάκων. Αν δεν είναι ίσα, επιστρέφει false. Αν δεν επιστρέψει από τον έλεγχο των μηκών, ψάχνει στη συνέχεια αν κάθε στοιχείο του tA είναι και στοιχείο του tB. Προσέξτε πως ο αντίστροφος έλεγχος είναι περιττός από τη στιγμή που τα μήκη είναι ίσα.

Ίδια είναι και η λογική της equals για πίνακες από Strings.

Η binarySearch εφαρμόζει τη λογική που αναπτύξαμε στην ενότητα 6.5.2.

Η swap κρατάει την πρώτη από τις τιμές που αντιμεταθέτει σε προσωρινή μεταβλητή, εκχωρεί τη δεύτερη τιμή στην πρώτη και στη θέση της δεύτερης τιμής εκχωρεί την τιμή της προσωρινής μεταβλητής.

Η sort υλοποιεί τον αλγόριθμο που παρατίθεται στον κώδικα 6.18. Η διαφορά της είναι πως την εύρεση του μικρότερου στοιχείου την αναθέτει στην idxMin.

7.9.6 Ταξινόμηση δισδιάστατου πίνακα με βάση το άθροισμα γραμμών

Να υλοποιηθεί συνάρτηση με διεπαφή public static void sortSubArraysBySum(int[][] t) που ταξινομεί τους πίνακες του t κατά αύξουσα αριθμητική σειρά του αθροίσματος των στοιχείων τους. Για παράδειγμα, ο πίνακας

```
{
  {100, 200, 300, 400},
  {1, 2, 3, 4, 5},
  {10, 20, 30, 40, 50, 60}
}
```

θα πρέπει να ταξινομηθεί ως

```
{
   {1, 2, 3, 4, 5},
   {10, 20, 30, 40, 50, 60},
   {100, 200, 300, 400}
}
```

καθώς 1+2=3+4+5 < 10+20+30+40+50+60 < 100+200+300+400΄

Λύση
Θα εφαρμόσουμε παρόμοια λογική με αυτήν της ταξινόμησης μονοδιάστατου πίνακα. Έτσι, θα χρειαστούμε μια συνάρτηση που εντοπίζει τον πίνακα-στοιχείο του t που έχει το μικρότερο άθροισμα.

```
static int idxOfMinSum(int[][] t, int startIdx) {
    int min = MyArrays.sum(t[startIdx]);
    int rVal = startIdx;
    for (int i = startIdx + 1; i < t.length; i++) {
        if (MyArrays.sum(t[i]) < min) {
            min = MyArrays.sum(t[i]);
            rVal = i;
        }
    }
    return rVal;
}
```

Κώδικας 7.26 *Εύρεση πίνακα στοιχείου με το μικρότερο άθροισμα*

Ο κώδικας 7.26 παρουσιάζει την idxOfMinSum που αρχίζοντας μετά από τη θέση startIdx εντοπίζει ποιος πίνακας-στοιχείο του t έχει το μικρότερο άθροισμα στοιχείων.
Έχοντας στη διάθεσή μας την idxOfMinSum, μπορούμε να υλοποιήσουμε την απαιτούμενη ταξινόμηση εφαρμόζοντας αντίστοιχη λογική με αυτήν της ταξινόμησης μονοδιάστατου πίνακα. Ο κώδικας 7.27 παρουσιάζει τη σχετική υλοποίηση:

```
public static void sortSubArraysBySum(int[][] t) { //Κώδικας 7.27
    for (int i = 0; i < t.length; i++) {
        int mIdx = idxOfMinSum(t, i);
        if (mIdx != i) {
            int[] local = t[i];
            t[i] = t[mIdx];
            t[mIdx] = local;
        }
    }
```

```
    }
```

Κώδικας 7.27 *Ταξινόμηση σύμφωνα με το άθροισμα των γραμμών δισδιάστατου πίνακα*

7.9.7 Διαχείριση ψηφιοσειρών

Να υλοποιηθεί η κλάση BitMaps που περιλαμβάνει τις λειτουργίες

```
public static int setBit(int i, int p)
```

Θέτει στο bit που βρίσκεται στη θέση p αντιγράφου του ακέραιου i την τιμή 1. Επιστρέφει τον παραγόμενο ακέραιο.

```
public static int resetBit(int i, int p)
```

Θέτει στο bit που βρίσκεται στη θέση p αντιγράφου του ακέραιου i την τιμή 0. Επιστρέφει τον παραγόμενο ακέραιο.

```
public static int getBit(int i, int p)
```

Επιστρέφει την τιμή του bit που βρίσκεται στη θέση p του ακέραιου i

```
public static int reverseBit(int i, int p)
```

Αντιστρέφει σε αντίγραφο του i το bit στη θέση p. Επιστρέφει τον παραγόμενο ακέραιο.

Λύση

Παρουσιάζουμε τον κώδικα της κλάσης BitMap ακολουθούμενο από τις αναγκαίες επεξηγήσεις.

```
public class BitMap { //Κώδικας 7.28

    public static int setBit(int i, int p) {
        int local = 1;
        local <<= p;
        return i | local;
    }

    public static int resetBit(int i, int p) {
        int local = setBit(0, p);
        local = ~local;
        return i & local;
    }

    public static int getBit(int i, int p) {
        int local = setBit(0, p);
        if ((local & i) == local) {
            return 1;
        }
        return 0;
    }

    public static int reverseBit(int i, int p) {
        if (getBit(i, p) == 1) {
            return resetBit(i, p);
        } else {
            return setBit(i, p);
        }
    }

    public static void main(String[] args) {
        System.out.println(setBit(1, 1));
        System.out.println(resetBit(3, 0));
        System.out.println(getBit(3, 1));
```

```
        System.out.println(getBit(3, 2));
        System.out.println(reverseBit(3,0));
    }
}
```

Κώδικας 7.28 *Διαχείριση ψηφιοσειρών*

Η συνάρτηση setBit, στην πρώτη γραμμή αρχικοποιεί τη μεταβλητή local στην τιμή 1. Επομένως, ο ψηφιοχάρτης της local γίνεται 00000000000000000000000000000001, δηλαδή 31 μηδενικά ακολουθούμενα από έναν άσο. Παρότι ο int έχει μήκος 32 bits, για απλοποίηση, στο εξής θα θεωρούμε ότι είναι μήκους 8 bits. Με αυτήν την υπόθεση, η local έχει το ακόλουθο περιεχόμενο 00000001. Στη συνέχεια εκτελείται διολίσθηση στα αριστερά για όσες θέσεις καθορίζει η παράμετρος p. Για παράδειγμα, θεωρήστε την κλήση setBit(1,1) στην πρώτη γραμμή της main. Η p εδώ έχει την τιμή 1. Επομένως, η local θα αποκτήσει περιεχόμενο 00000010. Στη συνέχεια γίνεται bitwise OR μεταξύ της παραμέτρου i και της local. Η παράμετρος i έχει τιμή 1, δηλαδή περιεχόμενο 00000001. Συνεπώς 00000010 | 00000001=00000011, δηλαδή 3 στο δεκαδικό σύστημα που είναι και η τιμή που εμφανίζεται από την println στην πρώτη γραμμή της main.

Με άλλα λόγια διαμορφώσαμε έναν ψηφιοχάρτη στη μεταβλητή local ο οποίος έχει όλα τα bits ίσα με το 0, εκτός από το bit που θέλουμε να κάνουμε 1. Στη συνέχεια, το bitwise OR μεταξύ της local και οποιουδήποτε ακέραιου, x, θα μετατρέψει σε 1 το bit του x που βρίσκεται στη θέση που έχει η local τιμή 1.

Παρόμοια εργαζόμαστε και για την resetBit. Ορίζουμε τη μεταβλητή local που έχει όλα τα bits 0 εκτός από το bit στη θέση p. Με bitwise not αντιστρέφουμε όλα τα bits της local οπότε σε όλες τις θέσεις έχει πλέον 1 εκτός από τη θέση p που έχει 0. Στη συνέχεια το bitwise and με την παράμετρο i, έχει ως αποτέλεσμα όλα τα bits να παραμείνουν ως έχουν εκτός από το bit στη θέση p που αποκτά την τιμή 0.

Η getBit δημιουργεί τον ακέραιο local που αρχικοποιείται ώστε ο ψηφιοχάρτης του να αποτελείται από 0 εκτός από τη θέση p. Επομένως, αν το bitwise and μεταξύ της local και της i επιστρέψει ακέραιο ίσο με την local αυτό σημαίνει πως το bit στη θέση p της i είναι 1 αλλιώς 0.

Τέλος η reverseBit είναι απλή καθώς βασίζεται στις getBit, resetBit και setBit και δεν κάνει απευθείας πράξεις σε επίπεδο ψηφίου.

7.10 Ασκήσεις προς λύση

1. Να αναπτύξετε συνάρτηση double abs(double n) που επιστρέφει την απόλυτη τιμή του n.

2. Να αναπτύξετε συνάρτηση που υπολογίζει και τυπώνει την τιμή της $f(x)=x^4+5x^3+2x^2+8x+1$ για τις τιμές του x=1..100, όπου x ακέραιος.

3. Να αναπτύξετε συνάρτηση που τυπώνει τους ν πρώτους όρους της ακολουθίας Fibonacci.

4. Να αναπτυχθεί συνάρτηση που επιστρέφει true αν η ακέραιη παράμετρός της είναι πρώτος αριθμός.

5. Να αναπτυχθεί συνάρτηση που τυπώνει τους πρώτους αριθμούς από το 1 έως το 100.

6. Να αναπτυχθεί συνάρτηση που τυπώνει τους αριθμούς από το 1 έως το 100 που δεν είναι πρώτοι.

7. Να αναπτυχθεί συνάρτηση που λαμβάνει μια boolean παράμετρο και επιστρέφει έναν πίνακα με τους χαρακτήρες της Αγγλικής, κεφαλαίους ή πεζούς, ανάλογα με την τιμή της παραμέτρου.

8. Να υλοποιηθεί Java application που λαμβάνει 3 παραμέτρους που αντιπροσωπεύουν μια εξίσωση 2ου βαθμού ($ax^2+bx+c=0$) και τυπώνει τη λύση της εφόσον υπάρχει ή διαφορετικά το μήνυμα δεν υπάρχει λύση, διακρίνουσα <0.

9. Να αναπτυχθεί συνάρτηση που επιστρέφει το άθροισμα των ψηφίων ενός ακεραίου.

10. Να αναπτυχθεί συνάρτηση με διεπαφή static int max(int…in) που λαμβάνει μια σειρά από ακέραιους και επιστρέφει τον μεγαλύτερο.

11. Να αναπτυχθεί συνάρτηση που λαμβάνει ως παράμετρο έναν πραγματικό αριθμό που αναπαριστά θερμοκρασία σε Celsius και επιστρέφει τη θερμοκρασία σε Fahrenheit.

12. Υπερφορτώστε την equals της άσκησης 7.8.5 ώστε να εξετάζει και πίνακες πραγματικών.

13. Αναπτύξτε συνάρτηση με διεπαφή static int[] resize(int[] t, int newSize) που επιστρέφει ένα αντίγραφο του t με μήκος newSize. Αν η τιμή της newSize είναι μικρότερη από το αρχικό μήκος του t, τότε ενδέχεται να απωλεσθούν δεδομένα από τον πίνακα.

14. Οποιοσδήποτε βαθμός μικρότερος του 5 αντιστοιχεί σε αποτυχία. Η βαθμολογία από 5 έως 6,4 αντιστοιχεί στον χαρακτηρισμό «καλώς», από 6,5 έως 8,4 στο «λίαν καλώς» και από 8,5 έως 10 στο «άριστα». Να αναπτύξετε συνάρτηση που λαμβάνει ως είσοδο έναν βαθμό από 0 έως 10 και επιστρέφει τον αντίστοιχο χαρακτηρισμό.

15. Να αναπτυχθεί εφαρμογή Java που παράγει 100 τυχαίους αριθμούς από το 1 έως το 40 και στη συνέχεια μετράει.

- Πόσοι από τους παραγόμενους αριθμούς είναι μοναδικοί.
- Πόσοι είναι άρτιοι και πόσοι περιττοί.
- Ποιοι αριθμοί από το 1 έως το 40 δεν υπάρχουν καθόλου στην παραγόμενη ακολουθία.
- Ποιος ή ποιοι αριθμοί εμφανίζονται περισσότερες φορές.

20. Να αναπτυχθεί συνάρτηση με διεπαφή static void reverse(char[] t) που αντιστρέφει τις θέσεις των στοιχείων του t, δηλαδή τοποθετεί το στοιχείο από την πρώτη στην τελευταία θέση του πίνακα, από τη δεύτερη στην προτελευταία, κοκ.

21. Να αναπτυχθεί συνάρτηση που επιστρέφει τον μέγιστο κοινό διαιρέτη ενός πλήθους ακέραιων.

22. Να αναπτυχθεί συνάρτηση που επιστρέφει το ελάχιστο κοινό πολλαπλάσιο ενός πλήθους ακέραιων.

23. Να αναπτύξετε συνάρτηση που λαμβάνει ως παράμετρο έναν πίνακα ακέραιων και επιστρέφει επίσης έναν πίνακα ακέραιων που αναπαριστά το σύνολο των ακέραιων του πίνακα εισόδου. Προσέξτε πως σε ένα σύνολο, ένα στοιχείο μπορεί να υπάρχει μόνο μία φορά.

24. Να αναπτύξετε συνάρτηση που υλοποιεί την ταξινόμηση φυσαλίδας σε πίνακα πραγματικών.

25. Να αναπτύξετε συνάρτηση που εξετάζει αν ένας πίνακας είναι ταξινομημένος. Η συνάρτηση να επιστρέφει 0 για αταξινόμητους πίνακες, -1 για ταξινομημένους κατά φθίνουσα σειρά και 1 για ταξινομημένους κατά αύξουσα σειρά.

26. Να αναπτύξετε συνάρτηση που συγχωνεύει δύο ταξινομημένους πίνακες σε έναν νέο πίνακα επίσης ταξινομημένο.

27. Να αναπτύξετε συνάρτηση που λαμβάνει ως παράμετρο δύο μονοδιάστατους πίνακες ακέραιων και επιστρέφει έναν πίνακα που περιέχει μόνο τα κοινά στοιχεία των παραμέτρων της έτσι ώστε κάθε κοινό στοιχείο να υπάρχει μόνο μία φορά στον πίνακα επιστροφής.

28. Να αναπτύξετε συνάρτηση που λαμβάνει ως παράμετρο δύο τετραγωνικούς πίνακες πραγματικών και επιστρέφει το άθροισμά τους. Σημειώστε πως τετραγωνικοί λέγονται οι πίνακες που έχουν ίδιο αριθμό γραμμών και στηλών.

29. Να αναπτύξετε συνάρτηση που ανακατανέμει τα στοιχεία ενός μονοδιάστατου πίνακα.

30. Να αναπτύξετε συνάρτηση που ανακατανέμει τα στοιχεία ενός δισδιάστατου πίνακα έτσι ώστε ένα τελικό στοιχείο να είναι πιθανό να βρεθεί σε οποιαδήποτε γραμμή και στήλη του πίνακα.

31. Ο τόκος ενός κεφαλαίου υπολογίζεται με βάση το επιτόκιο. Όταν ο τόκος μίας περιόδου προστίθεται στο κεφάλαιο, τότε ο τόκος της επόμενης περιόδου υπολογίζεται επί του αρχικού κεφαλαίου αυξημένου κατά τον τόκο της προηγούμενης περιόδου. Το φαινόμενο αυτό ονομάζεται ανατοκισμός. Να αναπτύξετε συνάρτηση που λαμβάνει ως παραμέτρους το κεφάλαιο, το επιτόκιο και τον αριθμό των περιόδων και υπολογίζει το σύνολο του τόκου.

32. Να αναπτυχθεί συνάρτηση που επιστρέφει την κύρια διαγώνιο τετραγωνικού πίνακα.

33. Να αναπτυχθεί συνάρτηση που επιστρέφει το Ελάχιστο Κοινό Πολλαπλάσιο μιας σειράς ακέραιων.

34. Να αναπτυχθεί συνάρτηση που επιστρέφει τον Μέγιστο Κοινό Διαιρέτη μιας σειράς ακέραιων.

35. Έστω nb ένας αριθμός εκφρασμένος σε θεσιακό αριθμητικό σύστημα με βάση b. Η μετατροπή του στο δεκαδικό σύστημα αρίθμησης γίνεται από τον τύπο

$$n_b = s_k \times b^k + s_{k-1} \times b^{k-1} + \ldots + s_1 \times b^1 + s_0 \times b^0 + s_{-1} \times b^{-1} + s_{-2} \times b^{-2} + \ldots + s_m \times b^m$$

Θεωρήστε ως θεσιακά αριθμητικά συστήματα μόνο αυτά που μας ενδιαφέρουν στην Πληροφορική, δηλαδή το δυαδικό, το οκταδικό και το δεκαεξαδικό.

36. Να υλοποιηθεί συνάρτηση που λαμβάνει δύο παραμέτρους τύπου String. Η πρώτη αναπαριστά έναν αριθμό που μπορεί να περιέχει ή όχι, υποδιαιρέσεις της μονάδας, π.χ. 101101.98, 25634.01, AB32.A ή 101101, 25634, AB32. Η δεύτερη αναπαριστά τη βάση του αριθμητικού συστήματος στο οποίο εκφράζεται ο αριθμός και μπορεί να λάβει τις τιμές 2, 8 ή 16. Η συνάρτηση επιστρέφει την τιμή του αριθμού στο δεκαδικό σύστημα ως double.

Καλέστε τη συνάρτηση και μετατρέψτε τους ακόλουθους αριθμούς στο δεκαδικό σύστημα.

```
110111102
65128
A2E3416
10110.0112
763.2348
A2E34.A16
BB135.1B16
```

Βρείτε κατάλληλο υπολογιστή μετατροπών στο διαδίκτυο και ελέγξτε τα αποτελέσματα.

Κεφάλαιο 8

Σύνοψη
Σε αυτήν την ενότητα παρουσιάζονται τα βασικά στάδια της ανάπτυξης εφαρμογών προσαρμοσμένα στις γνώσεις που έχει αποκομίσει ο αναγνώστης μέχρι αυτό το σημείο του βιβλίου. Ο στόχος είναι να διευκολυνθεί ο αρχάριος προγραμματιστής να υιοθετήσει έναν δομημένο τρόπο ανάλυσης, σχεδίασης και υλοποίησης τόσο των ασκήσεων του βιβλίου όσο και γενικότερων εφαρμογών. Προς αυτόν τον στόχο παρουσιάζονται οι έννοιες του δομημένου προγραμματισμού, τα στάδια της ανάπτυξης, η συλλογή απαιτήσεων, η ανάλυση και ο σχεδιασμός, η τεκμηρίωση και απολαθοποίηση του κώδικα καθώς και οι διαδικασίες και τα εργαλεία ελέγχου της ορθότητάς του.

Προαπαιτούμενη γνώση
Οι θεμελιώδεις τύποι και βασική γνώση του τύπου String. Βασική είσοδος και έξοδος. Η λειτουργία των τελεστών. Οι δομές επιλογής και οι επαναληπτικές δομές. Η διαχείριση των πινάκων. Οι συναρτήσεις, η έννοια και βασική διαχείριση των απροσδόκητων λαθών, οι προσδιοριστές προσπέλασης και η χρήση των πακέτων.

Λέξεις κλειδιά
Δομημένος προγραμματισμός, απαιτήσεις, ανάλυση, σχεδιασμός, τεκμηρίωση, απολαθοποίηση.

8 Ανάπτυξη εφαρμογών

Η ανάπτυξη των εφαρμογών δεν αρχίζει ούτε τελειώνει με τη συγγραφή του κώδικα σε κάποια γλώσσα προγραμματισμού. Αντίθετα, μια σειρά από διαδικασίες που λαμβάνουν χώρα πριν, μετά και κατά τη διάρκεια της συγγραφής του κώδικα διαδραματίζουν κρίσιμο ρόλο στην ανάπτυξη μιας ολοκληρωμένης εφαρμογής που απαντά αποτελεσματικά στις ανάγκες των χρηστών της. Στις διαδικασίες αυτές συμπεριλαμβάνονται κατ' ελάχιστο, η συλλογή και ανάλυση των απαιτήσεων, ο σχεδιασμός της εφαρμογής, η κωδικοποίηση και απολαθοποίηση, η τεκμηρίωση του κώδικα και ο έλεγχος της ορθότητάς του.

Αυτές τις διαδικασίες παρουσιάζει αυτή η ενότητα προσαρμοσμένες στις ανάγκες μιας εισαγωγής στον προγραμματισμό. Ο σκοπός της ενότητας δεν είναι να αντικαταστήσει τη μελέτη των διάφορων τεχνικών ανάλυσης και σχεδίασης λογισμικού. Εξάλλου η μελέτη αυτών των τεχνικών είναι έξω από την εμβέλεια αυτού του βιβλίου. Ωστόσο, κάποιες κατευθυντήριες γραμμές είναι χρήσιμες και μπορεί να βοηθήσουν στην αποτελεσματική ανάπτυξη ορθότερων και δομημένων εφαρμογών ήδη από το στάδιο της εισαγωγής στον Προγραμματισμό.

Με βάση αυτό το σκεπτικό, σε αυτήν την ενότητα παρουσιάζεται μια απλοποιημένη εκδοχή των διαδικασιών συλλογής και ανάλυσης απαιτήσεων και σχεδιασμού λογισμικού. Οι υπόλοιπες διαδικασίες, δηλαδή η υλοποίηση, η απολαθοποίηση, η τεκμηρίωση και ο έλεγχος ορθότητας του προγράμματος, παρουσιάζονται στο πλαίσιο της Java, αξιοποιούν εργαλεία που σχετίζονται άμεσα με την Java και εκτίθενται με μεγαλύτερη λεπτομέρεια. Δεν θα πρέπει όμως να θεωρηθεί ότι τα θέματα που πραγματεύεται το κεφάλαιο αυτό αναπτύσσονται σε πλήρη έκταση. Ο αναγνώστης που επιθυμεί να εμβαθύνει θα πρέπει να ανατρέξει σε σχετική βιβλιογραφία. Ας εξηγήσουμε όμως καταρχάς την έννοια του δομημένου προγράμματος

8.1 Δομημένος Προγραμματισμός

Οι στόχοι του Δομημένου Προγραμματισμού (Structured Programming) είναι να καταστήσει τα προγράμματα ευανάγνωστα, να περιορίσει τα λάθη σε αυτά και να επιταχύνει τον χρόνο ανάπτυξης. Το κύριο συστατικό του είναι η ρουτίνα, δηλαδή το γνωστό μας υποπρόγραμμα ή η συνάρτηση στο πλαίσιο της Java. Η ρουτίνα θα πρέπει να είναι αυτόνομη και να δίνει απάντηση σε ένα σαφώς ορισμένο πρόβλημα, ενώ ο συνδυασμός των ρουτίνων θα πρέπει να γίνεται με τρόπο που δομημένα ανακλά το πρόβλημα που αποτελεί το κίνητρο της ανάπτυξης.

Ο Δομημένος Προγραμματισμός προέκυψε τη δεκαετία του 1950 με την ανάπτυξη των γλωσσών Algol [1] και σύντομα επικράτησε τόσο στην ακαδημαϊκή κοινότητα όσο και στον χώρο των επαγγελματιών προγραμματιστών. Η τεχνολογία του προγραμματισμού εκείνη την εποχή περιλάμβανε περίπου τις έννοιες που έχουν παρουσιαστεί μέχρι εδώ σε αυτό το βιβλίο. Ωστόσο, ο δομημένος προγραμματισμός κάθε άλλο

παρά ξεπερασμένο μοντέλο θεωρείται. Αντίθετα αποτελεί θεμελιώδες μοντέλο προγραμματισμού, καθώς ενυπάρχει στα περισσότερα πιο σύγχρονα μοντέλα όπως ο Αντικειμενοστρεφής Προγραμματισμός ή ο Προγραμματισμός Πρακτόρων (Agent Oriented Programming).

Αρχικά το μοντέλο βασίστηκε στο θεώρημα του δομημένου προγράμματος (structured program theorem) [2] σύμφωνα με το οποίο κάθε πρόγραμμα μπορεί να αναπτυχθεί συνδυάζοντας ρουτίνες με τρεις μόνο βασικές δομές.

1. Σειριακή εκτέλεση, δηλαδή η μία ρουτίνα εκτελείται σειριακά μετά από μία άλλη
2. Επιλογή, δηλαδή ο έλεγχος μιας λογικής συνθήκης αποφασίζει ποια ρουτίνα θα εκτελεστεί
3. Επανάληψη, δηλαδή μια ή περισσότερες ρουτίνες που συνδυάζονται σειριακά ή με επιλογή μπορούν να επαναλαμβάνονται κάτω από τον έλεγχο μιας λογικής συνθήκης.

Κάποιοι συγγραφείς προσθέτουν και την αναδρομή, δηλαδή τη δομή κατά την οποία μια ρουτίνα καλεί τον εαυτό της κάτω από μία συνθήκη ελέγχου. Ωστόσο, η αναδρομή δύσκολα μπορεί να θεωρηθεί βασική δομή, καθώς τα προβλήματα στα οποία αξιοποιείται μπορούν να επιλυθούν και με χρήση των τριών προαναφερόμενων δομών.

Σύντομα, η αποδοχή του μοντέλου έλαβε καθολικές διαστάσεις. Χαρακτηριστικό δείγμα η εξαφάνιση εντολών goto από τα προγράμματα παγκοσμίως. Οι εντολές τύπου goto αποτελούν ανεξέλεγκτες μεταπηδήσεις (unconditional jumps) από ένα σημείο του προγράμματος σε ένα άλλο, οπότε παραβιάζουν το θεώρημα του δομημένου προγράμματος, ενώ στην πράξη αποδεικνυόταν πως συχνά αποτελούσαν πηγή λαθών.

Στην πορεία στο μοντέλο έγιναν κάποιες παραλλαγές με κυριότερες την προσθήκη διαχείρισης εξαιρέσεων (Exceptions) και την πρόωρη έξοδο (early exit) από επαναληπτικές δομές και ρουτίνες.

Ο μηχανισμός των εξαιρέσεων αναπτύχθηκε τη δεκαετία του 1960 [3] ως μηχανισμός διαχείρισης απροσδόκητων γεγονότων που είναι πιθανό να συμβούν κατά την εκτέλεση του προγράμματος, π.χ. ένα πρόγραμμα επιχειρεί να ανοίξει ένα αρχείο για να επεξεργαστεί τα δεδομένα του αλλά το αρχείο δεν βρίσκεται στην καθορισμένη θέση στον δίσκο. Οι εξαιρέσεις συνδυάζουν δεδομένα από δύο διαφορετικές περιοχές του κώδικα, από το εσωτερικό της συνάρτησης κατά την εκτέλεση της οποίας συμβαίνει το απροσδόκητο γεγονός και από την περιοχή του κώδικα που καλεί αυτήν τη συνάρτηση. Με αυτήν την έννοια, οι εξαιρέσεις δεν υπακούουν στις τρεις αρχές του θεωρήματος του δομημένου προγράμματος. Ωστόσο, είναι αναγκαία η χρησιμοποίησή τους και όλες οι σύγχρονες γλώσσες ενσωματώνουν κάποιο μηχανισμό εξαιρέσεων.

Μια συνέπεια του θεωρήματος του δομημένου προγράμματος είναι η λεγόμενη μοναδική έξοδος (single exit). Σύμφωνα με την αρχή της μοναδικής εξόδου, οι επαναληπτικές δομές και οι συναρτήσεις οφείλουν να τερματίζουν από ένα μόνο συγκεκριμένο σημείο. Σήμερα υπάρχουν διαφορετικές προσεγγίσεις επάνω σε αυτό το θέμα με κυριότερη την πρόωρη έξοδο (early exit). Με βάση την προσέγγιση αυτή, μια συνάρτηση μπορεί να ελέγξει κάποιες συνθήκες και να προχωρήσει σε έξοδο από οποιοδήποτε σημείο. Κάτι τέτοιο είναι ασφαλέστερο να γίνει σε ένα αρχικό στάδιο πριν η συνάρτηση προχωρήσει σε διαχείριση πόρων. Σε πολλές περιπτώσεις, με την πρόωρη έξοδο ο κώδικας γίνεται απλούστερος, ασφαλέστερος και πιο κατανοητός. Η πρόωρη έξοδος εφαρμόζεται σε πολλούς κώδικες όπου θεωρείται χρήσιμη σε αυτό το βιβλίο. Η πρόωρη έξοδος υποστηρίζεται από όλες σχεδόν τις σύγχρονες γλώσσες και για τις επαναληπτικές δομές με τη χρήση των λέξεων-κλειδιών continue και break.

Οι έννοιες αναγνωσιμότητα προγράμματος (code readability), επαναχρησιμοποίηση κώδικα (code reusability) και αφαιρετικότητα (abstraction) σχετίζονται στενά μεταξύ τους αλλά και με το υπόδειγμα του δομημένου προγραμματισμού. Ωστόσο, οι έννοιες αυτές συναντώνται και σε άλλα μοντέλα προγραμματισμού, όπως ο αντικειμενοστρεφής προγραμματισμός. Η συζήτηση εδώ όμως αναφέρεται στις συγκεκριμένες έννοιες και το ειδικό περιεχόμενο που αυτές προσλαμβάνουν στο επίπεδο του δομημένου προγραμματισμού.

Ας δούμε όμως τι σημαίνει αφαιρετικότητα και επαναχρησιμοποίηση κώδικα στην πράξη και στο πλαίσιο της Java. Ας υποθέσουμε πως έχουμε ένα πρόβλημα στα πλαίσια του οποίου χρειάζεται να ψάξουμε σε ένα πίνακα μήκους 10 να διαπιστώσουμε αν υπάρχει ένα στοιχείο ή όχι. Μία λύση είναι να παρεμβάλουμε εκεί όπου χρειάζεται έναν κώδικα που ψάχνει στον συγκεκριμένο πίνακα. Μια καλύτερη λύση όμως είναι να κάνουμε μια συνάρτηση που λαμβάνει παραμετρικά τον πίνακα και το υπό αναζήτηση στοιχείο και επιστρέφει κατά πόσο το στοιχείο περιλαμβάνεται στον πίνακα. Η συνάρτηση αυτή μπορεί να χρησιμοποιηθεί και αλλού όπου είναι χρήσιμη. Η συνάρτηση θα πρέπει να σχεδιαστεί με όσο το δυνατόν μεγαλύτερη αφαιρετικότητα, δηλαδή με όσο το δυνατόν μικρότερη σχέση με το συγκεκριμένο πρόβλημα που αποτέλεσε

κίνητρο για την ανάπτυξή της. Με άλλα λόγια, η συνάρτηση δεν θα πρέπει να αξιοποιεί το γεγονός ότι ο πίνακας στον οποίο θέλουμε να ψάξουμε έχει μήκος 10. Αντίθετα, θα πρέπει να γραφεί κατά τρόπο που να μπορούμε να την καλέσουμε για οποιονδήποτε πίνακα ανεξάρτητα από το μήκος του. Είναι προφανές πως η αφαιρετικότητα συμβάλλει στη δυνατότητα επαναχρησιμοποίησης του κώδικα.

Στο συγκεκριμένο πρόβλημα αναζήτησης μας ενδιαφέρει αν μια τιμή υπάρχει σε έναν πίνακα, επομένως, η συνάρτηση θα μπορούσε να είναι τύπου boolean. Παρόλα αυτά, μπορούμε να αυξήσουμε τη δυνατότητα επαναχρησιμοποίησης της συνάρτησης αν αυτή επιστρέφει ακέραιο ο οποίος στην περίπτωση που βρεθεί το στοιχείο μεταφέρει τη θέση του στον πίνακα, αλλιώς μια τιμή που δεν μπορεί να αποτελεί θέση σε πίνακα, π.χ. μια αρνητική τιμή. Έτσι το πρόγραμμα-πελάτης, δηλαδή η εφαρμογή που θα καλεί τη συνάρτησή μας, μπορεί να την καλέσει απλώς για να ελέγξει αν υπάρχει ή όχι το στοιχείο αλλά μπορεί εφόσον το χρειάζεται να αξιοποιήσει τη θέση του στοιχείου στον πίνακα.

Ας σημειώσουμε εδώ πως η έννοια της δόμησης δεν αφορά στενά την κωδικοποίηση σε μια γλώσσα προγραμματισμού αλλά σχετίζεται και με την ανάλυση (structured analysis) και με τον σχεδιασμό (structured design).

8.2 Τα Στάδια ανάπτυξης

Στην υλοποίηση των εφαρμογών θα διευκολυνθούμε αν ακολουθούμε τα βήματα:

Συλλογή απαιτήσεων (Requirements Gathering). Θα πρέπει να καταγράψουμε με όσο μεγαλύτερη ακρίβεια τις απαιτήσεις των δυνητικών χρηστών της εφαρμογής, δηλαδή τι πρέπει να κάνει η εφαρμογή. Η ανάπτυξη σεναρίων χρήσης (use cases) μπορεί να φανεί πολύ χρήσιμη. Τα σενάρια χρήσης περιγράφουν καταστάσεις κατά τις οποίες το υπό ανάπτυξη λογισμικό μπορεί να είναι χρήσιμο ή υπό μία διαφορετική έννοια, περιγράφουν την ανταπόκριση του λογισμικού σε πιθανές εισόδους του χρήστη. Αποφάσεις σχετικά με το πώς θα επιτευχθούν οι απαιτήσεις ή ποια εργαλεία λογισμικού θα χρησιμοποιηθούν για τον σκοπό αυτό δεν περιλαμβάνονται σε αυτό το στάδιο.

Ανάλυση Απαιτήσεων (Requirements Analysis). Η ανάλυση των απαιτήσεων έχει ως σκοπό να αντιστοιχίσει τις απαιτήσεις των χρηστών σε μια σειρά σαφώς προσδιορισμένων εργασιών του υπό ανάπτυξη συστήματος.

Σχεδιασμός του Συστήματος (System Design). Ο στόχος εδώ είναι ο σχεδιασμός των αυτόνομων ρουτίνων και της σύνθεσής τους με σκοπό την υλοποίηση των εργασιών του συστήματος, όπως αυτές καθορίστηκαν στο στάδιο της ανάλυσης.

Υλοποίηση (Implementation). Εδώ περνάμε στην υλοποίηση των ρουτίνων που εντοπίστηκαν στον σχεδιασμό, δηλαδή στην κωδικοποίησή τους σε ένα πλαίσιο προγραμματισμού.

Έλεγχος (test). Τα λάθη στους κώδικες είναι συχνό και ως έναν βαθμό αναπόφευκτο πρόβλημα. Σε αυτό το στάδιο, θα πρέπει να ελέγξουμε συστηματικά την εκτέλεση και τα αποτελέσματα της εφαρμογής μας με σκοπό να εντοπίσουμε πιθανά προβλήματα.

Επειδή καμία εφαρμογή δεν παράγεται πλήρως ολοκληρωμένη, συχνά τα παραπάνω βήματα επαναλαμβάνονται είτε γιατί αποκαλύπτονται αδύνατα σημεία είτε γιατί προστίθενται επιπλέον απαιτήσεις. Έτσι μιλάμε για κυκλική ανάπτυξη (cyclical development) ή για κύκλο ζωής ανάπτυξης εφαρμογών (Application development life cycle). Συχνά μάλιστα, στο περιβάλλον του επαγγελματικού προγραμματισμού συμπεριλαμβάνονται στα στάδια ανάπτυξης, η διανομή του λογισμικού (deployment) και η παρακολούθησή του στον πελάτη (monitoring).

Στη συνέχεια, βασιζόμενοι σε ένα απλό παράδειγμα προσπαθούμε να δείξουμε πώς μπορούμε να εφαρμόζουμε τα στάδια της ανάπτυξης στο επίπεδο των απλών εφαρμογών αυτού του βιβλίου ώστε από την αρχή να μάθουμε να δομούμε τις εφαρμογές όσο το δυνατόν αποτελεσματικότερα.

8.3 Συλλογή απαιτήσεων

Η συλλογή των απαιτήσεων (requirements gathering) είναι από μόνη της απαιτητική δουλειά. Σε γενικές γραμμές, οι απαιτήσεις χωρίζονται σε δύο κατηγορίες, στις απαιτήσεις του χρήστη (user requirements) και στις λειτουργικές απαιτήσεις (functional requirements).

Στις απαιτήσεις του χρήστη συμπεριλαμβάνονται απαιτήσεις τόσο των άμεσων χρηστών της εφαρμογής όσο και απαιτήσεις άλλων παραγόντων όπως η αγορά, η επιχείρηση για την οποία σχεδιάζεται το

λογισμικό, κλπ. Για παράδειγμα, ο χρήστης θέλει να μπορεί να πλοηγείται σε λίστα προϊόντων και να βρίσκει εύκολα το προϊόν που επιθυμεί. Αυτό είναι μια απαίτηση του χρήστη. Όταν ο χρήστης πλοηγείται σε λίστα προϊόντων, θα πρέπει τα προϊόντα στη λίστα να ταξινομούνται ανάλογα με το τι έχει ήδη αγοράσει ο συγκεκριμένος χρήστης. Αυτό είναι μια λειτουργική απαίτηση. Με άλλα λόγια, οι απαιτήσεις του χρήστη περιγράφουν τι θέλει ο χρήστης από την εφαρμογή, ενώ οι λειτουργικές απαιτήσεις περιγράφουν τι κάνει η εφαρμογή.

Δεν πρόκειται να επεκταθούμε άλλο στο μεγάλο και σημαντικό θέμα της συλλογής απαιτήσεων. Ο αναγνώστης που ενδιαφέρεται για περισσότερες λεπτομέρειες μπορεί να διαβάσει το [4]. Στη συνέχεια, θα υποθέσουμε πως το κείμενο που ακολουθεί περιγράφει τις απαιτήσεις για μια εφαρμογή. Πρόκειται στην ουσία για ένα παράδειγμα αναφορικά με το οποίο θα παρουσιαστούν και οι υπόλοιπες διεργασίες του κύκλου της ανάπτυξης.

Παράδειγμα απαιτήσεων Χρήστη

Τρεις φίλοι, ο John, ο Jim και ο Jack θα παίξουν ένα παιχνίδι ζαριών. Σύμφωνα με τους κανόνες του παιχνιδιού, κατά τη διάρκεια μιας παρτίδας, οι παίκτες ρίχνουν εναλλάξ το ζάρι 10 φορές και μετράνε πόσες φορές έφερε ο καθένας τους άρτια ζαριά. Νικητής είναι εκείνος που έφερε τις περισσότερες άρτιες ζαριές. Αν όμως δύο ή περισσότεροι παίκτες έχουν ισοβαθμήσει, η παρτίδα επαναλαμβάνεται για όλους. Η επανάληψη συνεχίζεται μέχρις ότου να επικρατήσει ένας μοναδικός νικητής.
Οι συγκεκριμένοι παίκτες όμως έχουν τις ιδιομορφίες τους. Ο John είναι ένας τίμιος παίκτης με την έννοια πως κάθε φορά που πετάει το ζάρι έχει την ίδια πιθανότητα να φέρει οποιοδήποτε αποτέλεσμα από 1 έως 6. Από την άλλη μεριά, ο Jim είναι τυχερός, καθώς έχει αποδειχτεί πως όταν πετάει το ζάρι έχει 15% πιθανότητα για κάθε ένα από τα αποτελέσματα 1, 3 και 5, 18% για κάθε ένα από τα αποτελέσματα 2 ή 4 και 19% για να φέρει 6. Επομένως, ο Jim φέρνει άρτια ζαριά με πιθανότητα 55% και περιττή ζαριά με πιθανότητα 45%.
Αντίθετα, ο Jack είναι άτυχος, καθώς φέρνει 2, 4, 6 με πιθανότητα 15% για κάθε ένα από αυτά τα αποτελέσματα, ενώ 1 και 3 φέρνει με πιθανότητα 18% και 5 φέρνει με πιθανότητα 19%. Επομένως, ο Jack φέρνει άρτια ζαριά με πιθανότητα 45% και περιττή ζαριά με πιθανότητα 55%.
Να υλοποιηθεί εφαρμογή που προσομοιώνει ένα παιχνίδι ζαριών μεταξύ των τριών παικτών και αναφέρει τον νικητή.

8.4 Σχεδιασμός

Γενικά υπάρχουν δύο βασικές προσεγγίσεις στον σχεδιασμό ενός προγράμματος. Η από πάνω προς τα κάτω (top-down) και η από κάτω προς τα επάνω (bottom-up).

Σύμφωνα με την top-down, θα πρέπει αρχικά να σχεδιάσουμε την εφαρμογή καταρχάς στο υψηλότερο επίπεδο. Για παράδειγμα, μιλώντας με όρους Java, θα πρέπει πρώτα να σχεδιάσουμε την main. Σε αυτό το επίπεδο πρέπει να γνωρίζουμε τι θα κάνουν οι συναρτήσεις που καλούνται από την main χωρίς όμως να έχουμε καθορίσει τις λεπτομέρειες υλοποίησης τους. Στη συνέχεια, με τον ίδιο τρόπο σχεδιάζουμε κάθε συνάρτηση που καλείται από την main, δηλαδή καταρχάς χρησιμοποιούμε κλήσεις συναρτήσεων που δεν έχουμε ακόμη υλοποιήσει. Έτσι, σύμφωνα με την προσέγγιση top-down, σε κάθε βήμα αποσυνθέτουμε ένα πρόβλημα σε άλλα απλούστερα προβλήματα. Εξ' αυτού του λόγου, η μέθοδος αυτή συχνά ονομάζεται και σταδιακή υλοποίηση (stepwise refinement).

Αντίθετα, η προσέγγιση bottom-up ξεκινά τον σχεδιασμό στο χαμηλότερο επίπεδο και σε κάθε βήμα συνθέτει επιμέρους στοιχεία που οδηγούν σε υψηλότερο επίπεδο, δηλαδή σε σύστημα σταδιακά πιο πολύπλοκο.

Κάθε μια από τις μεθοδολογίες αυτές έχει πλεονεκτήματα και μειονεκτήματα. Η top-down προσέγγιση αργεί σχετικά να παράγει ολοκληρωμένες συναρτήσεις με αποτέλεσμα να καθυστερεί η διαδικασία ελέγχου και άρα η ολοκλήρωση του έργου. Η bottom-up είναι ένα στοίχημα, καθώς είναι δύσκολο να καθοριστούν με ακρίβεια οι λεπτομερείς ανάγκες πριν καθοριστεί το πλαίσιο στο οποίο αυτές θα αξιοποιηθούν. Συχνά, στην πράξη χρησιμοποιείται ένας συνδυασμός των δύο προσεγγίσεων. Για το παράδειγμα του παιχνιδιού ζαριών χρησιμοποιούμε την top-down προσέγγιση.

Αναλύοντας λοιπόν το κείμενο των απαιτήσεων του χρήστη, βλέπουμε πως εκείνο που ζητά είναι η προσομοίωση ενός παιχνιδιού ζαριών και η ανακήρυξη του νικητή. Ένα παιχνίδι όμως ζαριών συνίσταται στην επανάληψη μιας παρτίδας έως ότου ανακηρυχθεί μοναδικός νικητής.

Με βάση αυτήν την πληροφορία, εύκολα μπορούμε να αρχίσουμε τον σχεδιασμό χρησιμοποιώντας διαγράμματα ροής ή ψευδοκώδικα.

```
game:
     do {
          scores=oneLot;
     } while withdraw(scores);
     return winner(scores);
```

Κώδικας 8.1 *Ψευδοκώδικας για το παιχνίδι ζαριών*

Ο κώδικας 8.1 παρουσιάζει σε μορφή ψευδοκώδικα έναν σχεδιασμό για την υλοποίηση της κύριας απαίτησης του χρήστη. Πράγματι, εδώ ορίζεται μια ρουτίνα που ονομάζεται game και συνίσταται στην επανάληψη μιας παρτίδας ζαριών (oneLot) έως ότου προκύψει μοναδικός νικητής. Σε αυτό το στάδιο δεν μας ενδιαφέρει πώς ακριβώς θα επιτυγχάνει τους στόχους της η oneLot αλλά μόνο πώς αυτή είναι μια διαδικασία που προσομοιώνει μια παρτίδα και επιστρέφει τα σκορ των παικτών. Το ίδιο ισχύει και για την withdraw. Δεν μας αφορούν οι λεπτομέρειες υλοποίησής της αλλά μόνο ότι ανιχνεύει ενδεχόμενη ισοπαλία με βάση μια λίστα από σκορ. Τέλος, η winner λαμβάνει μια λίστα από σκορ τέτοια ώστε σε αυτήν υπάρχει ένα μόνο μέγιστο σκορ και επιστρέφει τον νικητή. Υποθέτουμε πως η winner λαμβάνει λίστα από σκορ στην οποία υπάρχει μία μόνο μέγιστη τιμή, καθώς η λίστα έρχεται από το παιχνίδι το οποίο φτάνει σε αυτό το σημείο μόνο εφόσον δεν ανιχνευθούν ισοπαλίες.

Θα πρέπει τώρα να συνεχίσουμε την ανάλυση και τον σχεδιασμό στο αμέσως επόμενο επίπεδο, δηλαδή να παράγουμε ψευδοκώδικα για τις oneLot, withdraw και winner.

Ας σχεδιάσουμε λοιπόν την παρτίδα, oneLot. Αυτή συνίσταται σε 10 ζαριές που ρίχνουν εναλλάξ, οι τρεις παίκτες. Όμως η ρίψη κάθε παίκτη έχει διαφορετικά χαρακτηριστικά. Επομένως, η ρίψη της ζαριάς θα πρέπει να παραμετροποιηθεί ως προς την κατανομή πιθανοτήτων στη βάση των οποίων υπολογίζεται η ρίψη κάθε παίκτη. Επιπλέον, θα ήταν καλό να παραμετροποιήσουμε την παρτίδα ως προς τον αριθμό ρίψεων. Με αυτόν τον τρόπο θα ανταποκριθούμε πιο εύκολα σε περίπτωση που θα γίνει κάποια σχετική αλλαγή στον αριθμό ρίψεων της παρτίδας. Στον κώδικα 8.2 χρησιμοποιούμε την noOfRolls αντί για τη σταθερά 10.

```
oneLot:
     for (noOfRolls times) {

          rslt = roll(Fair);
          if (rslt is even)
               update score for JOHN;

          rslt = roll(lucky);
          if (rslt is even)
               update score for JIM;

          rslt = roll(unlucky);
          if (rslt is even)
               update score for JACK;

     }
     return scores;
```

Κώδικας 8.2 *Ψευδοκώδικας για μια παρτίδα ζαριών*

Ο ψευδοκώδικας στον κώδικα 8.2 δίνει επαναληπτικά τη δυνατότητα σε κάθε παίκτη να πραγματοποιήσει μία ρίψη και αν το αποτέλεσμα είναι άρτιος ενημερώνει το αντίστοιχο σκορ. Σε αυτό το βήμα προκύπτει η ανάγκη να δούμε σε δεύτερο επίπεδο τον σχεδιασμό της roll και της ενημέρωσης του σκορ κάθε παίκτη.

Συνεχίζοντας όμως σε αυτό το επίπεδο μας μένει να σχεδιάσουμε τον έλεγχο της ισοπαλίας και τον εντοπισμό του νικητή. Πρόκειται για δύο πολύ απλές διαδικασίες. Ο εντοπισμός του νικητή πρέπει απλώς να ψάξει σε μια λίστα με σκορ και να επιστρέψει το μεγαλύτερο, οπότε δεν χρειάζεται να επεκταθούμε

περισσότερο σε αυτό. Ο έλεγχος της ισοπαλίας πρέπει να ψάξει σε μια λίστα με σκορ να διαπιστώσει εάν υπάρχουν περισσότερες από μία μέγιστες τιμές. Επομένως

```
withdraw:
     return countOccurances(score, maxScore)>1
```

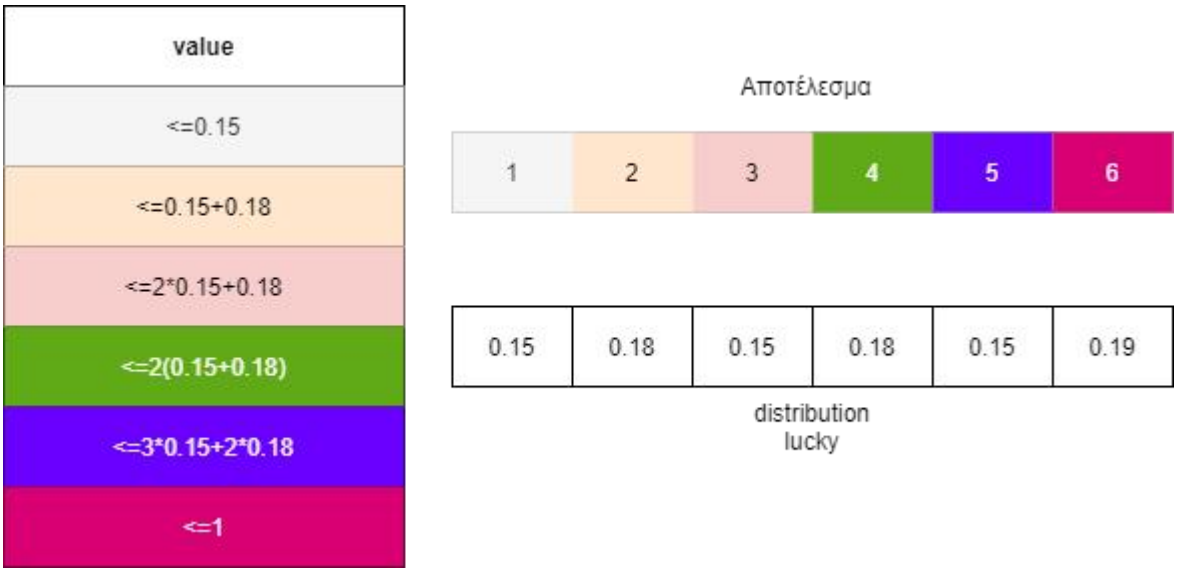
Κώδικας 8.3 *Ανίχνευση ισοπαλίας*

Στο επόμενο επίπεδο θα πρέπει να σχεδιάσουμε καταρχάς τη διαδικασία ρίψης, roll. Η ενημέρωση του σκορ του χρήστη είναι προφανώς πολύ απλή διαδικασία και δεν χρειάζεται περαιτέρω ανάλυση σε αυτό το στάδιο. Καθώς έχουμε τρεις roll οι οποίες επιστρέφουν έναν αριθμό από το 1 έως το 6 με διαφορετική κατανομή πιθανοτήτων για κάθε δυνατό αποτέλεσμα, μπορούμε να σχεδιάσουμε μια διαδικασία παραμετροποιημένη ως προς την κατανομή πιθανοτήτων. Επομένως, θεωρούμε πως η roll λαμβάνει μια λίστα που περιέχει έξι πιθανότητες, μία για κάθε δυνατό αποτέλεσμα μιας ρίψης. Στον ψευδοκώδικα 8.4, η λίστα αυτή ονομάζεται distribution.

```
roll:
     value = Random value from 0 to 1;
     sum = 0;
     rslt=1;
     for (each percentage in distribution) {
          sum += percentage;
          if (value <= sum)
               return rslt;
          rslt++;
     }
}
```

Κώδικας 8.4 *Η διαδικασία προσομοίωσης της ζαριάς*

Το σκεπτικό του κώδικα 8.4 έχει ως εξής: Παράγουμε έναν τυχαίο αριθμό από 0 έως 1, την value. Αν η value είναι μικρότερη ή ίση από την πιθανότητα για αποτέλεσμα 1, επιστρέφουμε 1. Διαφορετικά, αν η value είναι μεγαλύτερη από την πιθανότητα για 1 αλλά μικρότερη από το άθροισμα πιθανοτήτων για 1 και 2, τότε επιστρέφουμε 2. Την ίδια λογική ακολουθούμε για όλα τα δυνατά αποτελέσματα. Με αυτόν τον τρόπο πετυχαίνουμε κάθε αποτέλεσμα με βάση την πιθανότητα που δίνεται στο distribution.



Σχήμα 8.1 *Υπολογισμός της ρίψης με βάση την κατανομή πιθανοτήτων για κάθε δυνατό αποτέλεσμα.*

Στο σχήμα 8.1 οι τιμές της value αντιστοιχίζονται χρωματικά με τα δυνατά αποτελέσματα για την τυχερή κατανομή. Αν η τιμή της value είναι μικρότερη ή ίση του 0.15, η roll επιστρέφει 1. Επομένως, το 1

επιστρέφεται με πιθανότητα 15%. Αν η τιμή της value είναι μικρότερη ή ίση του 0.15+0.18 αλλά δεν είναι μικρότερη ή ίση από 0.15, περίπτωση κατά την οποία η roll θα είχε ήδη επιστρέψει 1, η τιμή επιστροφής είναι 2. Επομένως, η πιθανότητα να επιστρέψει η roll 2 είναι ίση με την πιθανότητα η value να έχει τιμή μικρότερη ή ίση από 0.15+0.18 μείον την πιθανότητα να έχει η value τιμή 0.15. Άρα, η πιθανότητα να επιστρέψει 2 είναι 0.18, δηλαδή σύμφωνη με την κατανομή. Με τον ίδιο τρόπο ερμηνεύεται και ο υπολογισμός των υπόλοιπων τιμών.

8.5 Υλοποίηση

Έχοντας ολοκληρώσει τον σχεδιασμό, μπορούμε να προχωρήσουμε στην υλοποίηση. Στο στάδιο αυτό πρέπει να φροντίσουμε να αξιοποιήσουμε όλες τις δυνατότητες που μας προσφέρει η γλώσσα υλοποίησης, δηλαδή στην προκειμένη περίπτωση, η Java, ώστε ο κώδικάς μας να είναι ευανάγνωστος, απαλλαγμένος κατά το δυνατό από λάθη και εύκολα συντηρήσιμος. Στη συνέχεια παρουσιάζουμε πρώτα την υλοποίηση και μετά την σχολιάζουμε.

```
import java.util.Random;

public class ADiceGame {

    static final double EPSILON = 0.00001;
    private static final Random gen = new Random();
    static final double[] fair = {1/6d, 1/6d, 1/6d, 1/6d, 1/6d, 1/6d};
    static final double[] lucky = {15d/100, 18d/100, 15d/100, 18d/100, 15d/100,
19d/100};
    static final double[] unlucky = {18d/100, 15d/100, 18d/100, 15d/100, 19d/100,
15d/100};
    private static final int JOHN = 0;
    private static final int JIM = 1;
    private static final int JACK = 2;

    public static boolean approximateEquals(double d1, double d2) {
        return Math.abs(d1 - d2) < EPSILON;
    }

    public static double sum(double[] array) {
        double sum = 0;
        for (double d : array) {
            sum += d;
        }
        return sum;
    }

    public static int roll(double[] distribution) {
        if (distribution.length != 6 || !approximateEquals(sum(distribution), 1d))
{
            throw new RuntimeException();
        }
        double value = gen.nextDouble();
        double sum = 0;
        int rslt = 1;
        for (double percentage : distribution) {
            sum += percentage;
            if (value <= sum) {
                return rslt;
            }
            rslt++;
        }
        return 0;
    }
```

```
public static int max(int[] array) {
    int idx = 0;
    int max = array[0];
    for (int i = 1; i < array.length; i++) {
        if (array[i] > max) {
            max = array[i];
            idx = i;
        }
    }
    return idx;
}

public static int cntOccurances(int[] array, int value) {
    int cnt = 0;
    for (int i = 0; i < array.length; i++) {
        if (value == array[i]) {
            cnt++;
        }
    }
    return cnt;
}

public static boolean withdraw(int[] score) {
    return cntOccurances(score, score[max(score)]) > 1;
}

private static int[] aLot(int noOfRolls) {
    int[] score = new int[3];
    for (int i = 0; i < noOfRolls; i++) {
        int rslt = roll(fair);
        if (rslt % 2 == 0) {
            score[JOHN]++;
        }

        rslt = roll(lucky);
        if (rslt % 2 == 0) {
            score[JIM]++;
        }

        rslt = roll(unlucky);
        if (rslt % 2 == 0) {
            score[JACK]++;
        }
    }
    return score;
}

public static int game() {
    final int NOOFROLLS = 10;
    int[] score;

    do {
        score = aLot(NOOFROLLS);
    } while (withdraw(score));
    return max(score);
}

private static String winnerName(int winnerId) {

    switch (winnerId) {
```

```
            default:
                throw new RuntimeException();
            case JOHN:
                return "John";
            case JIM:
                return "Jim";
            case JACK:
                return "Jack";
        }
    }

    public static void main(String[] args) {
        int winnerId = game();
        System.out.println("winner is " + winnerName(winnerId));
    }
}
```

Κώδικας 8.5 *Εφαρμογή – Ένα παιχνίδι ζαριών*

Ο κώδικας 8.5 παρουσιάζει την υλοποίηση του παιχνιδιού ζαριών. Στην main καλείται η game που επιστρέφει τον νικητή. Στη συνέχεια, τυπώνεται το όνομά του με τη βοήθεια της winnerName.
Η game υλοποιεί ένα παιχνίδι με συγκεκριμένα χαρακτηριστικά παρτίδας. Έτσι δηλώνει ως τοπική σταθερά τον αριθμό ρίψεων κάθε παρτίδας. Στη συνέχεια, σύμφωνα με τον σχεδιασμό καλεί την aLot μέσα σε επαναληπτική διαδικασία που ελέγχεται από την withdraw. Σε κάθε κλήση, η aLot επιστρέφει έναν πίνακα με τα σκορ των παικτών. Τα σκορ ελέγχει η withdraw. Αν η withdraw δεν ανιχνεύσει ισοπαλία, τερματίζεται ο βρόχος do και επιστρέφεται η θέση του πίνακα των σκορ με τη μεγαλύτερη τιμή. Εκεί βρίσκεται ο νικητής.

Η aLot λαμβάνει ως παράμετρο τον αριθμό των ρίψεων ανά παρτίδα και παίκτη. Δηλώνει τον πίνακα score όπου αποθηκεύονται οι ζαριές με άρτιο αποτέλεσμα για κάθε παίκτη κατά τη διάρκεια της παρτίδας. Προκειμένου να αντιστοιχίζεται κάθε παίκτης ευκολότερα, με περισσότερη ασφάλεια και να είναι το πρόγραμμα ευανάγνωστο, έχουν δηλωθεί οι σταθερές JOHN, JIM και JACK ως 0, 1 και 2. Έτσι γίνεται μια αντιστοιχία μεταξύ των θέσεων στον πίνακα των σκορ με τα ονόματα των παικτών. Μέσα στον βρόγχο for της aLot κάθε παίκτης ρίχνει τη ζαριά του με τη βοήθεια της roll και αν το αποτέλεσμα είναι άρτιο, ενημερώνεται το σκορ του.

Η withdraw, σύμφωνα με τον σχεδιασμό μετράει πόσες φορές στον πίνακα των σκορ υπάρχει το μέγιστο σκορ. Επιστρέφει true μόνο αν βρει περισσότερες από μια μέγιστες τιμές. Την απαιτούμενη μέτρηση υλοποιεί η cntOccurances με τη βοήθεια της max που εντοπίζει τη μέγιστη τιμή.

Η roll λαμβάνει ως παράμετρο έναν πίνακα από πραγματικούς που αναπαριστά την κατανομή των πιθανοτήτων για κάθε δυνατό αποτέλεσμα. Επειδή στο συγκεκριμένο παιχνίδι έχουμε τρεις πιθανές κατανομές, έχουν οριστεί τρεις πίνακες, ο fair, ο lucky και ο unlucky που αναπαριστούν τις τρεις κατανομές. Κάθε πίνακας έχει 6 στοιχεία, καθώς δίνει μια πιθανότητα για κάθε ένα από τα έξι πιθανά αποτελέσματα. Επειδή ρίχνοντας το ζάρι η πιθανότητα να έρθει ένα από τα έξι πιθανά αποτελέσματα είναι 100%, το άθροισμα των στοιχείων κάθε ενός πίνακα πρέπει να είναι 1. Τους ελέγχους αυτούς κάνει η roll. Αν δεν επαληθευτούν, παράγει εξαίρεση. Στη συνέχεια εφαρμόζει τον αλγόριθμο σύμφωνα με τον σχεδιασμό και επιστρέφει το αποτέλεσμα.

Η sum και η approximateEquals είναι προφανείς και έχουν ήδη συζητηθεί.

Προσέξτε πως κάποια μέλη της ADiceGame είναι δημόσια, άλλα ιδιωτικά και άλλα έχουν πρόσβαση πακέτου. Πιο συγκεκριμένα, η EPSILON και οι κατανομές έχουν πρόσβαση πακέτου. Όπως θα δούμε, οι μεταβλητές αυτές θα μας είναι χρήσιμες κατά την ανάπτυξη του ελέγχου του κώδικα και για αυτόν τον λόγο τους δώσαμε πρόσβαση πακέτου, διαφορετικά θα μπορούσαμε να σκεφτούμε να τις ορίσουμε ως ιδιωτικές, καθώς σχετίζονται πολύ στενά με τη συγκεκριμένη εφαρμογή. Η γεννήτρια τυχαίων αριθμών gen είναι ένα αντικείμενο που χρησιμοποιείται στη συγκεκριμένη κλάση για την παραγωγή τυχαίων αριθμών και δεν υπάρχει λόγος να δώσουμε πρόσβαση σε αυτήν έξω από την κλάση. Σε περίπτωση που άλλες κλάσεις επιθυμούν να παράγουν τυχαίους αριθμούς μπορούν να δημιουργήσουν δικές τους γεννήτριες. Τα ονόματα των παικτών αφορούν αποκλειστικά το συγκεκριμένο παιχνίδι και δεν χρειάζεται να έχει πρόσβαση σε αυτά κανείς άλλος.

Επίσης, ως ιδιωτικές έχουν δηλωθεί οι aLOt, game και winnerName. Αυτή η επιλογή οφείλεται στο γεγονός ότι οι τρεις αυτές συναρτήσεις είναι πολύ εξειδικευμένες και αφορούν αποκλειστικά τη συγκεκριμένη εφαρμογή.

Τα υπόλοιπα μέλη της κλάσης έχουν οριστεί ως δημόσια κυρίως με την έννοια πως παρουσιάζουν τέτοια χρησιμότητα ώστε είναι πιθανό να αξιοποιηθούν και από τρίτους.

8.6 Τεκμηρίωση

Η τεκμηρίωση των προγραμμάτων της Java γίνεται με το ειδικό εργαλείο παραγωγής τεκμηρίωσης JavaDoc [3, 4]. Όλες οι προκαθορισμένες κλάσεις της Java είναι τεκμηριωμένες με το JavaDoc. Αν πάτε σε έναν κώδικα στο NetBeans σε ένα σημείο που αναφέρεται το όνομα μιας προκαθορισμένης κλάσης, π.χ. String και πατήσετε Alt+F1, θα διαπιστώσετε πως θα ανοίξει η τεκμηρίωση της κλάσης String[7]. Εκεί μπορείτε να βρείτε τεκμηρίωση για κάθε μέλος της κλάσης.

Η συγγραφή της τεκμηρίωσης κάθε κλάσης βασίζεται σε ειδικά σχόλια, τα επονομαζόμενα σχόλια τεκμηρίωσης (doc comments). Σχόλιο τεκμηρίωσης θεωρείται οποιοδήποτε κείμενο βρίσκεται μεταξύ των χαρακτήρων /** που σηματοδοτούν την αρχή του σχολίου και τους χαρακτήρες */ που σηματοδοτούν το τέλος του. Επομένως, υπάρχει διαφορά από τα απλά σχόλια που περιλαμβάνονται μεταξύ των χαρακτήρων /* και */. Τα μεν σχόλια τεκμηρίωσης απαιτούν δύο αστερίσκους μετά την πλάγια κάθετο ενώ τα απλά ένα. Τα απλά σχόλια δεν λαμβάνονται υπόψη από το JavaDoc.

Τα σχόλια τεκμηρίωσης τοποθετούνται πριν τη δήλωση οποιασδήποτε κλάσης ή οποιουδήποτε μέλους κλάσης. Ωστόσο, υπάρχει η δυνατότητα να παράγουμε τεκμηρίωση που αφορά ένα πακέτο, δηλαδή ένα σύνολο κλάσεων ή και για ένα σύνολο πακέτων [5].

Η τεκμηρίωση JavaDoc αφορά κυρίως τα δημόσια μέλη των κλάσεων. Παρότι το εργαλείο προσφέρει τη δυνατότητα τεκμηρίωσης και των μη δημόσιων μελών, η χρησιμότητά της αμφισβητείται από τη μεγάλη πλειοψηφία των επαγγελματιών του χώρου. Αντί της τεκμηρίωσης JavaDoc για τα μη δημόσια μέλη είναι προτιμητέα τα απλά σχόλια με βάση το σκεπτικό πως τα δημόσια μέλη θα χρησιμοποιηθούν από τρίτους επομένως μόνο η τεκμηρίωση των δημόσιων μελών έχει ενδιαφέρον προς τρίτους.

Κατά τη συγγραφή τεκμηρίωσης μπορούμε να χρησιμοποιήσουμε μια σειρά από tags για να προσδιορίσουμε τον τύπο της πληροφορίας που περιγράφουμε. Πριν τη χρήση των tags, μπορεί να προστεθεί μια περιγραφή του υπό τεκμηρίωση αντικειμένου. Παρουσιάζουμε εδώ τα πολύ βασικά tags. Πλήρης λίστα των διαθέσιμων tags μπορεί να βρεθεί στην [6].

Βασικά tags
author
see
param
return
throws

Πίνακας 8.1 *Βασικά tags JavaDoc*

Στο tag author αναφέρεται προφανώς το όνομα του συγγραφέα ή των συγγραφέων του κώδικα. Το tag εισάγεται αυτόματα από τα περισσότερα IDE και από το NetBeans. Ωστόσο, δεν συμπεριλαμβάνεται στο παραγόμενο JavaDoc.

Το tag see χρησιμοποιείται όταν η τεκμηρίωση αναφέρεται σε κάποια άλλη κλάση ή μέλος κλάσης από αυτήν στην οποία ανήκει το tag. Το tag δημιουργεί έναν σύνδεσμο στο αναφερόμενο αντικείμενο.

Το tag param χρησιμοποιείται για την περιγραφή παραμέτρου συνάρτησης.

Το tag return για την περιγραφή της επιστρεφόμενης τιμής από συνάρτηση.

---

[7] Αν με Alt+F1 δεν ανοίξει αυτόματα η τεκμηρίωση της κλάσης String, μεταβείτε στο NetBeans στο μενού Tools/Java Platforms. Στο παράθυρο που θα ανοίξει, επιλέξτε το (Default) JDK. Σύμφωνα με τις οδηγίες αυτού του βιβλίου πρέπει να είναι το JDK 17. Από τις καρτέλες δεξιά, επιλέξτε την καρτέλα JavaDoc και μεταβάλλετε το URL https://docs.oracle.com/en/Java/Javase/16/docs/api σε https://docs.oracle.com/en/Java/Javase/16/docs/api/Java.base.

Τέλος, το tag throws για την περιγραφή τυχόν εξαιρέσεων που μπορεί να παράγονται από συνάρτηση. Στη συνέχεια στον κώδικα 8.6 παρουσιάζουμε τον κώδικα του παιχνιδιού ζαριών με απλά σχόλια και με σχόλια τεκμηρίωσης ανάλογα με τον χαρακτηρισμό προσπέλασης συναρτήσεων και μεταβλητών.

```
import java.util.Random;

  /**
   *Η κλάση προσομοιώνει το ακόλουθο παιχνίδι ζαριών. Τρεις
   *φίλοι, ο John, ο Jim και ο Jack παίζουν ένα παιχνίδι
   *ζαριών. Σύμφωνα με τους κανόνες του παιχνιδιού, κατά τη
   *διάρκεια μιας παρτίδας, οι παίκτες ρίχνουν εναλλάξ το ζάρι
   *10 φορές και μετράνε πόσες φορές έφερε ο καθένας τους άρτια
   *ζαριά.
   * Νικητής είναι εκείνος που έφερε τις περισσότερες άρτιες
   *ζαριές. Αν όμως 2 ή περισσότεροι παίκτες έχουν
   *ισοβαθμήσει, η παρτίδα επαναλαμβάνεται για όλους.
   * Η επανάληψη συνεχίζεται μέχρις ότου να επικρατήσει ένας
   *μοναδικός νικητής. Οι συγκεκριμένοι παίκτες όμως έχουν τις
   *ιδιομορφίες τους. Ο John είναι ένας τίμιος παίκτης με την
   *έννοια πως κάθε φορά που πετάει το ζάρι έχει την ίδια
   *πιθανότητα να φέρει οποιοδήποτε αποτέλεσμα από 1 έως 6. Από
   *την άλλη μεριά, ο Jim είναι τυχερός καθώς έχει αποδειχτεί
   *πως όταν πετάει το ζάρι έχει 15% πιθανότητα για κάθε ένα
   *από τα αποτελέσματα 1, 3 και 5, 18% για κάθε ένα από τα
   *αποτελέσματα 2 ή 4 και 19% για να φέρει 6. Επομένως, ο Jim
   *φέρνει άρτια ζαριά με πιθανότητα 55% και περιττή ζαριά με
   *πιθανότητα 45%. Αντίθετα, ο Jack είναι άτυχος καθώς φέρνει
   *2, 4, 6 με πιθανότητα 15% για κάθε ένα από αυτά τα
   *αποτελέσματα, ενώ 1 και 3 φέρνει με πιθανότητα 18% και 5
   *φέρνει με πιθανότητα 19%. Επομένως, ο Jack φέρνει άρτια
   *ζαριά με πιθανότητα 45% και περιττή ζαριά * με πιθανότητα
   *55%.
   *
   * @author Lefteris Moussiades
   */
public class ADiceGame {

    /**
     * Χρησιμοποιείται απο τον έλεγχο ισότητας πραγματικών
     *
     * @see approximateEquals
     */
    static final double EPSILON = 0.00001;
    private static final Random gen = new Random();
    static final double[] fair = {1/6d, 1/6d, 1/6d, 1/6d, 1/6d, 1/6d};
    static final double[] lucky = {15d/100, 18d/100, 15d/100, 18d/100, 15d/100,
19d/100};
    static final double[] unlucky = {18d/100, 15d/100, 18d/100, 15d/100, 19d/100,
15d/100};
    private static final int JOHN = 0;
    private static final int JIM = 1;
    private static final int JACK = 2;

    /**
     * Συγκρίνει προσεγγιστικά δύο πραγματικούς τύπου double για έλεγχο
     * ισότητας
     *
     * @param d1
     * @param d2
     * @return Επιστρέφει true αν η απόλυτη τιμή της διαφορας d1-d2 είναι
     * μικρότερη απο την τιμή της EPSILON
```

```
     * @see EPSILON
     */
    public static boolean approximateEquals(double d1, double d2) {
        return Math.abs(d1 - d2) < EPSILON;
    }

    /**
     * Υπολογίζει το άθροισμα των στοιχείων της παραμέτρου array
     *
     * @param array
     * @return Το άθροισμα των στοιχείων της array
     */
    public static double sum(double[] array) {
        double sum = 0;
        for (double d : array) {
            sum += d;
        }
        return sum;
    }

    /**
     * Προσομοιώνει τη ρίψη ζαριού
     *
     * @param distribution Πίνακας πραγματικών. Κατά την κλήση της μεθόδου, η
     * distribution πρέπει να περιλαμβάνει 6 στοιχεία με άθροισμα ίσο με 1.
     * @return Έναν αριθμό απο 1 έως 6 ανάλογα με την πιθανότητα που δίνεται
     * για κάθε τιμή μέσω της παραμέτρου distribution
     * @throws RuntimeException
     */
    public static int roll(double[] distribution) {
        if (distribution.length != 6 || !approximateEquals(sum(distribution), 1d))
{
            throw new RuntimeException();
        }
        double value = gen.nextDouble();
        double sum = 0;
        int rslt = 1;
        for (double percentage : distribution) {
            sum += percentage;
            if (value <= sum) {
                return rslt;
            }
            rslt++;
        }
        return 0;
    }

    /**
     * Εντοπίζει τη μέγιστη τιμή του πίνακα array
     *
     * @param array Ο πίνακας του οποίου το μέγιστο στοιχείο θέλουμε να
     * βρούμε
     * @return Επιστρέφει τη θέση της πρώτης απο τις μέγιστες τιμές του
     * πίνακα
     * array
     */
    public static int max(int[] array) {
        int idx = 0;
        int max = array[0];
        for (int i = 1; i < array.length; i++) {
            if (array[i] > max) {
```

```
            max = array[i];
            idx = i;
        }
    }
    return idx;
}

/**
 * Υπολογίζει πόσες φορές υπάρχει μια τιμή σε έναν πίνακα
 *
 * @param array Ο πίνακας στον οποίο θα μετρήσουμε πόσες φορές υπάρχει η
 * value
 * @param value Η τιμή που θα μετρήσουμε πόσες φορές υπάρχει στον array
 * @return Επιστρέφει το πλήθος των στοιχείων με τιμή ίση με value στον
 * πίνακα array
 */
public static int cntOccurances(int[] array, int value) {
    int cnt = 0;
    for (int i = 0; i < array.length; i++) {
        if (value == array[i]) {
            cnt++;
        }
    }
    return cnt;
}

private static boolean withdraw(int[] score) {
    return cntOccurances(score, score[max(score)]) > 1;
}

/*
Η aLot  λαμβάνει ως παράμετρο τον αριθμό των ρίψεων ανα παρτίδα και
παίκτη.
Δηλώνει τον πίνακα score όπου αποθηκεύονται οι ζαριές με άρτιο αποτέλεσμα
για κάθε παίκτη κατά τη διάρκεια της παρτίδας.
Προκειμένου να αντιστοιχίζεται κάθε παίκτης ευκολότερα, με περισσότερη
ασφάλεια και να είναι το πρόγραμμα ευανάγνωστο,
έχουν δηλωθεί οι σταθερές JOHN, JIM και JACK ως 0, 1 και 2. Έτσι γίνεται
μια αντιστοιχία μεταξύ των θέσεων στον πίνακα των σκορ
με τα ονόματα των παικτών.
Μέσα στον βρόγχο for της aLot κάθε παίκτης ρίχνει τη ζαριά του με τη
βοήθεια της roll και αν το αποτέλεσμα είναι άρτιο ενημερώνεται
το σκορ του.
*/
private static int[] aLot(int noOfRolls) {
    int[] score = new int[3];
    for (int i = 0; i < noOfRolls; i++) {
        int rslt = roll(fair);
        if (rslt % 2 == 0) {
            score[JOHN]++;
        }

        rslt = roll(lucky);
        if (rslt % 2 == 0) {
            score[JIM]++;
        }

        rslt = roll(unlucky);
        if (rslt % 2 == 0) {
            score[JACK]++;
        }
```

```
        }
        return score;
    }

    /*
    Η game υλοποιεί ένα παιχνίδι με συγκεκριμένα χαρακτηριστικά παρτίδας.
    Έτσι δηλώνει ως τοπική σταθερά τον αριθμό ρίψεων κάθε παρτίδας.
    Στη συνέχεια, σύμφωνα με τον σχεδιασμό καλεί την aLot μέσα σε
    επαναληπτική διαδικασία που ελέγχεται από την withdraw.
    Σε κάθε κλήση, η aLot επιστρέφει έναν πίνακα με τα σκορ των παικτών.
    Τα σκορ ελέγχει η withdraw. Αν η withdraw δεν ανιχνεύσει ισοπαλία,
    τερματίζεται ο βρόχος do και επιστρέφεται η θέση του πίνακα
    των σκορ με τη μεγαλύτερη τιμή. Εκεί βρίσκεται ο νικητής.
     */
    private static int game() {
        final int NOOFROLLS = 10;
        int[] score;

        do {
            score = aLot(NOOFROLLS);
        } while (withdraw(score));
        return max(score);
    }

    private static String winnerName(int winnerId) {

        switch (winnerId) {
            default:
                throw new RuntimeException();
            case JOHN:
                return "John";
            case JIM:
                return "Jim";
            case JACK:
                return "Jack";
        }
    }

    public static void main(String[] args) {
        int winnerId = game();
        System.out.println("winner is " + winnerName(winnerId));

    }
}
```

Κώδικας 8.6 *Ο κώδικας του παιχνιδιού ζαριών σχολιασμένος*

Μετά την ολοκλήρωση σχολιασμού του κώδικα, μεταβείτε στην περιοχή Projects, κάντε δεξί κλικ στο όνομα του project στο οποίο έχετε σχολιάσει τις κλάσεις και επιλέξτε Generate Javadoc. Η τεκμηρίωση του κώδικα σε μορφή HTML θα ανοίξει στον φυλλομετρητή του συστήματος. Επιπλέον, με δεξί κλικ επάνω σε κάθε κλάση και επιλογή Tools/Analyze Javadoc μπορείτε να λάβετε μια αναφορά για τυχόν παραλείψεις στην τεκμηρίωση της κλάσης.

8.7 Απολαθοποίηση

Η απολαθοποίηση (debugging) είναι μια σημαντική λειτουργία χρήσιμη τόσο κατά το στάδιο της υλοποίησης όσο και κατά το στάδιο του ελέγχου όπου ενδέχεται να προκύψουν προβλήματα και να κληθούν οι προγραμματιστές να εντοπίσουν την εστία τους μέσα στον κώδικα και να τα θεραπεύσουν. Η απολαθοποίηση παρέχει τη δυνατότητα να εκτελέσουμε τον κώδικα ελεγχόμενα, βήμα προς βήμα και σε κάθε βήμα να ελέγχουμε τις τιμές των μεταβλητών και των εκφράσεων.

Θα δούμε τη λειτουργία του απολαθοποιητή στο NetBeans με ένα παράδειγμα.

```
import gr.ihu.cs.lmous.OOJ.k7.v1.StaticFunctions;
import java.util.Arrays;

public class Debugging {

    static int[] resize(int[] in, int newSize) {
        int[] rVal=new int[newSize];
        for (int i=0; i<in.length && i<rVal.length; i++)
            rVal[i]=in[i];
        return rVal;
    }

    static int[] createSet(int... elements) {
        int[] set=new int[elements.length];
        int setIdx=0;
        for (int element:elements) {
            int idx=StaticFunctions.sequentialSearch(set, element);
            if (idx<0)
                set[setIdx++]=element;
        }
        resize(set, setIdx);
        return set;
    }

    static int[] union(int[] set1, int[] set2) {
        int[] union=new int[set1.length+set2.length];
        System.arraycopy(set1, 0, union, 0, set1.length);
        System.arraycopy(set2, 0, union, set1.length, set2.length);
        return createSet(union);
    }

    public static void main(String[] args) {
        int[] set1=createSet(2,4,6,3), set2=createSet(3,5,7,2);
        int[] union=union(set1, set2);
        System.out.println(Arrays.toString(union));
    }
}
```

Κώδικας 8.7 *Κώδικας προς απολαθοποίηση*

Ο κώδικας 8.7 είναι μια εφαρμογή διαχείρισης συνόλων ακέραιων. Κάθε στοιχείο ενός συνόλου είναι μοναδικό, δηλαδή σε ένα σύνολο δεν μπορεί να υπάρχει δύο η περισσότερες φορές το ίδιο στοιχείο. Η συνάρτηση createSet λαμβάνει ως παράμετρο μια σειρά ακέραιων και επιστρέφει ένα σύνολο ακέραιων με τη μορφή πίνακα. Για να το πετύχει αυτό, εισάγει στον πίνακα που επιστρέφει μια τιμή από τις πραγματικές παραμέτρους μόνο εφόσον αυτή δεν υπάρχει ήδη. Μάλιστα πριν επιστρέψει καλεί την resize ώστε να ρυθμίσει το μέγεθος του πίνακα που επιστρέφει ανάλογα με το πλήθος των στοιχείων του συνόλου. Η συνάρτηση union λαμβάνει ως παραμέτρους δύο πίνακες που αντιπροσωπεύουν σύνολα και επιστρέφει την ένωσή τους. Στην main δημιουργούμε δύο σύνολα, υπολογίσουμε την ένωσή τους και την τυπώνουμε.
Αν τρέξετε τον κώδικα θα δείτε πώς η έξοδός του είναι

```
[2, 4, 6, 3, 5, 7, 0, 0]
```

Το σύνολο-ένωση όμως που αναμένουμε είναι το [2, 4, 6, 3, 5, 7]. Επομένως κάποιο λάθος υπάρχει στον κώδικα. Μπορείτε να το βρείτε; Αξίζει να προσπαθήσετε καταρχάς απλώς διαβάζοντας τον κώδικα. Αν ωστόσο δεν τα καταφέρετε, ο απολαθοποιητής είναι εδώ για να σας βοηθήσει.

Μεταβείτε στο NetBeans και δημιουργήστε την κλάση Debugging. Στην πρώτη γραμμή της main, κάντε κλικ στην γκρίζα κάθετη αριθμημένη λωρίδα στα αριστερά του κώδικα. Η γραμμή θα αποκτήσει ένα ροζ φόντο όπως φαίνεται στην εικόνα 8.1.

```
39      public static void main(String[] args) {
            int[] set1=createSet(2,4,6,3), set2=createSet(3,5,7,2);
41          int[] union=union(set1, set2);
42          System.out.println(Arrays.toString(union));
43      }
```

Εικόνα 8.1 *Breakpoint στην πρώτη γραμμή της main*

Έχετε τοποθετήσει ένα Breakpoint (σημείο διακοπής). Μπορείτε τώρα να εκτελέσετε τον κώδικα κατά τρόπο ώστε να διακοπεί η ροή του όταν συναντήσει το Breakpoint. Ανάλογα με τις ανάγκες σας μπορείτε να τοποθετήσετε και άλλα Breakpoint.
Μεταβείτε στην κλάση στο πλαίσιο Projects. Με δεξί κλικ πάνω στην κλάση ανοίγει ένα μενού, επιλέξτε Debug File. Θα μεταβείτε στην κατάσταση που δείχνει η εικόνα 8.2.

```
39      public static void main(String[] args) {
            int[] set1=createSet(2,4,6,3), set2=createSet(3,5,7,2);
41          int[] union=union(set1, set2);
42          System.out.println(Arrays.toString(union));
43      }
```

Εικόνα 8.2 *Διακοπή της ροής στο Breakpoint*

Η εφαρμογή εκτελείται αλλά έχει σταματήσει στην πρώτη γραμμή της main, εκεί δηλαδή που συνάντησε το πρώτο Breakpoint. Στη συνέχεια έχουμε έλεγχο στη ροή του προγράμματος με τη βοήθεια των εντολών προς τον απολαθοποιητή. Στη γραμμή εργαλείων, κάτω από το κεντρικό μενού βλέπετε την εργαλειοθήκη με τις διαθέσιμες εντολές προς τον απολαθοποιητή.

Εικόνα 8.3 *Η εργαλειοθήκη του απολαθοποιητή.*

Με αναφορά στην αρίθμηση της εικόνας 8.3, οι διαθέσιμες εντολές προς τον απολαθοποιητή έχουν ως εξής:

Finish Debugger Session: Τερματίζεται η λειτουργία του απολαθοποιητή.

Pause: Προσωρινή παύση της εκτέλεσης του υπό απολαθοποίηση προγράμματος.

Continue: Συνέχιση της εκτέλεσης του υπό απολαθοποίηση προγράμματος.

Step Over: Εκτέλεση της τρέχουσας συνάρτησης.

Step Over Expression: Εκτέλεση της τρέχουσας έκφρασης.

Step Into: Είσοδος στην τρέχουσα συνάρτηση.

Step Out: Έξοδος από την τρέχουσα συνάρτηση.

Run to cursor: Εκτέλεση του προγράμματος μέχρι του σημείου που έχουμε τοποθετήσει τον cursor στον κώδικα.

Καθώς το πρόγραμμα έχει μπει σε κατάσταση απολαθοποίησης, στο πλαίσιο κάτω από τον κώδικα, εκεί όπου βρίσκεται το παράθυρο Output, έχουν ανοίξει δύο ακόμη παράθυρα, το Variables και το Breakpoints. Στο παράθυρο Breakpoints μπορούμε να βλέπουμε και να διαχειριζόμαστε τα Breakpoints που τοποθετούμε στον κώδικα και στο παράθυρο Variables βλέπουμε τις μεταβλητές που είναι εντός εμβέλειας

ανάλογα με το σημείο του προγράμματος στο οποίο βρισκόμαστε. Για παράδειγμα, στην τρέχουσα κατάσταση, μπορούμε να δούμε πως η παράμετρος args έχει μήκος 0, κλήθηκε επομένως η εφαρμογή χωρίς παραμέτρους της main.

Πατήστε το εικονίδιο Step Over. Η πρώτη γραμμή έχει τρέξει. Ο απολαθοποιητής περιμένει στην επόμενη γραμμή, ενώ παράλληλα στο παράθυρο Variables έχουν εμφανιστεί οι μεταβλητές set1 και set2.

Μπορούμε να ανοίξουμε τις μεταβλητές για να δούμε τα περιεχόμενά τους. Είναι αυτά που αναμένουμε; Πράγματι, μια πρώτη επιθεώρηση μας δίνει τις αναμενόμενες τιμές τόσο για την set1 όσο και για την set2. Δεδομένου ότι έχουμε ένα πολύ μικρό πρόγραμμα και είδαμε πως η πρώτη γραμμή εκτελείται με το σωστό αποτέλεσμα, λογικό είναι να υποθέσουμε πως κάτι δεν πάει καλά στη δεύτερη γραμμή όπου εκτελείται η union. Για αυτόν τον λόγο θα επιθεωρήσουμε πιο στενά την union.

Πατήστε Step Into, η ροή του κώδικα θα σας μεταφέρει στην πρώτη γραμμή της union. Αν προχωρήσουμε βήμα προς βήμα με Step Over μέχρι την τελευταία γραμμή της union, δεν θα συναντήσουμε κάποιο πρόβλημα. Επομένως θα υποπτευθούμε πως κάτι μπορεί να συμβαίνει στην τελευταία γραμμή της union όπου καλείται η createSet. Μα την createSet την έχουμε ήδη δοκιμάσει και φαίνεται να πήγε καλά. Ας μπούμε καλύτερα μέσα με Step Into να δούμε με μεγαλύτερη λεπτομέρεια.

Με Step Over στις γραμμές της create και έλεγχο των διαθέσιμων μεταβλητών διαπιστώνουμε πως το πρόβλημα είναι πως το μέγεθος του πίνακα set δεν μεταβάλλεται μετά την εκτέλεση της resize. Ενώ ο πίνακας set δημιουργήθηκε αρχικά με μέγεθος το άθροισμα των μεγεθών των δύο συνόλων που ενώνονται, μετά την resize το μέγεθός του αναμένεται μειωμένο κατά δύο καθώς τα στοιχεία 2 και 3 υπάρχουν και στα δύο σύνολα.

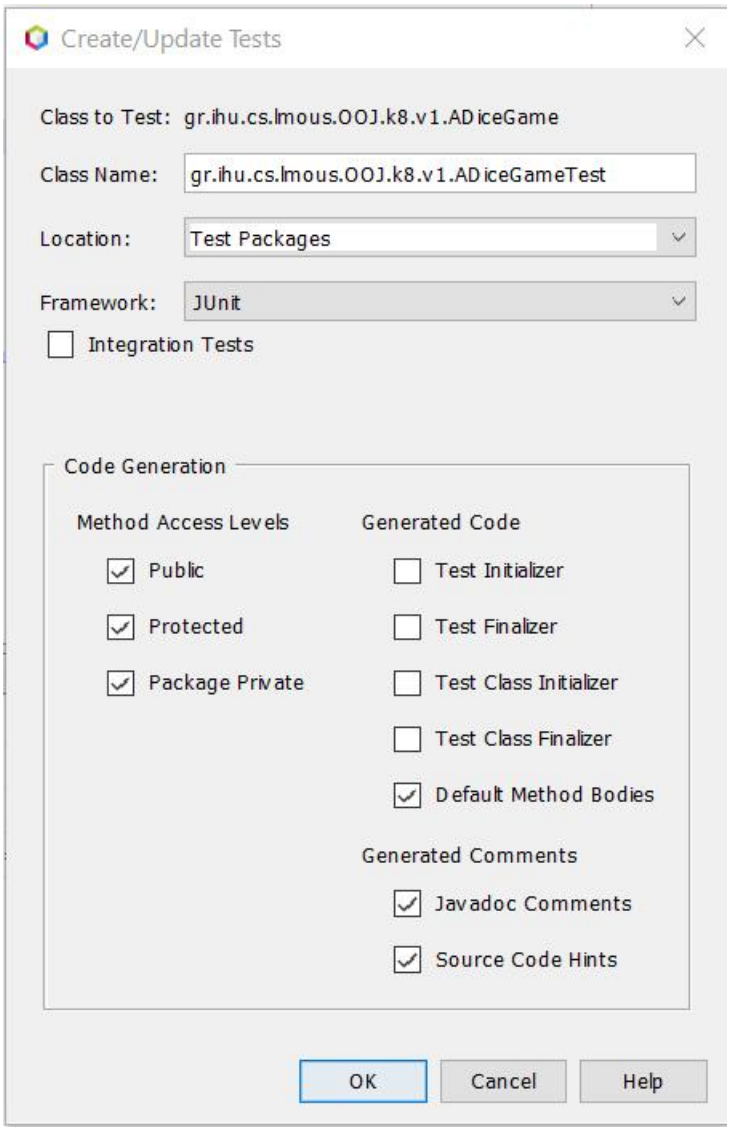
Μετά τη διαπίστωση αυτή, παρατηρήστε πως η resize δεν μεταβάλλει το μήκος του πίνακα που περνάει ως παράμετρος σε αυτήν. Αντίθετα, δημιουργεί και επιστρέφει ένα αντίγραφο της παραμέτρου με το σωστό μήκος. Εφόσον επιθυμούμε να αλλάξουμε το μέγεθος του πίνακα set, θα πρέπει να εκχωρήσουμε την τιμή που επιστρέφει η resize στη μεταβλητή set. Επομένως αλλάξτε τη γραμμή resize(set, setIdx) σε set=resize(set, setIdx), τερματίστε τον απολαθοποιητή και ξανατρέξτε την κλάση. Θα διαπιστώσετε πως η ένωση των συνόλων εμφανίζεται σωστά αυτήν τη φορά.

8.8 Έλεγχος

Ο έλεγχος (test) αποτελεί ένα απαραίτητο στοιχείο της ανάπτυξης. Είναι σημαντικό να κατανοήσουμε πως ο έλεγχος δεν γίνεται αποκλειστικά μετά την ολοκλήρωση της υλοποίησης. Αντίθετα, ξεκινά παράλληλα με την υλοποίηση και διεξάγεται καθόλη τη διάρκειά της.

Ο έλεγχος επιτυγχάνεται με τη συγγραφή και εκτέλεση κατάλληλων συναρτήσεων. Θα πρέπει δηλαδή για κάθε μια μη ιδιωτική συνάρτηση της εφαρμογής να αναπτύξουμε μια αντίστοιχη συνάρτηση ελέγχου. Θα δούμε περισσότερες λεπτομέρειες στην πράξη αναπτύσσοντας κώδικα ελέγχου για την κλάση ADiceGame.

Στο παράθυρο Projects, μεταβείτε στην 1η έκδοση της κλάσης και πατήστε δεξί κλικ. Στο παράθυρο που θα ανοίξει επιλέξτε Tools/Create Tests. Θα ανοίξει το παράθυρο που φαίνεται στην εικόνα 8.4.



Εικόνα 8.4 *Παράμετροι δημιουργίας test*

Στην ενότητα Code Generation, θέστε τις επιλογές όπως φαίνονται στην εικόνα 8.4 και πατήστε OK. Θα διαπιστώσετε πως δημιουργήθηκαν δύο νέοι φάκελοι κάτω από το project. Οι φάκελοι Test Packages και Test Libraries. Κάντε δεξί κλικ στον φάκελο Test Libraries και στο μενού που θα ανοίξει επιλέξτε Add Library. Θα ανοίξει τότε ένα παράθυρο με τις διαθέσιμες βιβλιοθήκες. Εντοπίστε το Junit 4.12, επιλέξτε το και πατήστε Add Library. Ανοίξτε τον φάκελο Test Libraries και διαγράψτε (remove) τις υπόλοιπες βιβλιοθήκες πλην της Junit 4.12 και της Hamcrest 1.3. Αν δεν βλέπετε την Hamcrest, προσθέστε την όπως προσθέσαμε την Junit 4.12.

Μεταβείτε στον φάκελο Test Packages, εντοπίστε το ADiceGameTest και διαγράψτε το. Τέλος μεταβείτε και πάλι στην 1η έκδοση της κλάσης, πατήστε δεξί κλικ και δημιουργήστε εκ νέου το ADiceGameTest. Η κλάση θα ανοίξει αυτόματα στην περιοχή του κώδικα. Περιηγηθείτε στον κώδικα της κλάσης και παρατηρήστε πως για κάθε μη ιδιωτική συνάρτηση της κλάσης AdiceGame, η κλάση AdiceGameTest περιλαμβάνει μια συνάρτηση ελέγχου. Για παράδειγμα, για την συνάρτηση approximateEquals περιλαμβάνει την TestApproximateEquals, για την sum την TestSum, κοκ. Θα πρέπει να διορθώσουμε κατάλληλα τις συναρτήσεις ελέγχου ώστε να επιτελούν το έργο τους.

Αντικαταστήστε την ADiceGameTest που παρήχθη αυτόματα με την ADiceGameTest του κώδικα 8.9 όπου έχουν γίνει οι απαιτούμενες αλλαγές.

```
import java.util.Random;
import org.junit.Test;
import static org.junit.Assert.*;
```

```java
/**
 *
 * @author Lefteris Moussiades <lmous@cs.ihu.gr>
 */
public class ADiceGameTest {

    /**
     * Test of approximateEquals method, of class ADiceGame.
     */
    @Test
    public void testApproximateEquals() {
        System.out.println("approximateEquals");
        double d1 = 0.0;
        double d2 = ADiceGame.EPSILON;
        boolean result = ADiceGame.approximateEquals(d1, d2);
        assertEquals(false, result);
        d2 = ADiceGame.EPSILON / 2;
        assertEquals(true, ADiceGame.approximateEquals(d1, d2));
    }

    /**
     * Test of sum method, of class ADiceGame.
     */
    @Test
    public void testSum() {
        System.out.println("sum");
        double[] array = ADiceGame.fair;
        double result = ADiceGame.sum(array);
        assertEquals(1d, result, ADiceGame.EPSILON);
        array = ADiceGame.lucky;
        result = ADiceGame.sum(array);
        assertEquals(1d, result, ADiceGame.EPSILON);
        array = ADiceGame.unlucky;
        result = ADiceGame.sum(array);
        assertEquals(1d, result, ADiceGame.EPSILON);
    }

    /**
     * Test of roll method, of class ADiceGame.
     */
    @Test
    public void testRoll() {
        System.out.println("roll");
        double[] distribution;
        int sumFair = 0, sumLucky = 0, sumUnlucky = 0, rslt;
        for (int i = 0; i < 10000; i++) {
            rslt = ADiceGame.roll(ADiceGame.fair);
            if (rslt < 1 || rslt > 6) {
                fail("roll result " + rslt);
            }
            sumFair += rslt % 2 == 0 ? 1 : 0;
            rslt = ADiceGame.roll(ADiceGame.lucky);
            if (rslt < 1 || rslt > 6) {
                fail("roll result " + rslt);
            }
            sumLucky += rslt % 2 == 0 ? 1 : 0;
            rslt = ADiceGame.roll(ADiceGame.unlucky);
            if (rslt < 1 || rslt > 6) {
                fail("roll result " + rslt);
            }
```

```
            sumUnlucky += rslt % 2 == 0 ? 1 : 0;
        }
        System.out.println(sumLucky/10000d+" "+sumFair/10000d+"
"+sumUnlucky/10000d);
        assertTrue(sumLucky > sumFair && sumFair > sumUnlucky);
    }

    /**
     * Test of max method, of class ADiceGame.
     */
    public static void shuffle(int[] t) {
        Random r = new Random();

        for (int i = 0; i < t.length; i++) {
            int rIdx = r.nextInt(t.length);
            int tmp = t[i];
            t[i] = t[rIdx];
            t[rIdx] = tmp;
        }
    }

    private int[] constructArray(int initialValue) {
        int[] rVal = new int[10];
        for (int i = 0; i < 10; i++) {
            rVal[i] = initialValue + i;
        }
        shuffle(rVal);
        return rVal;
    }

    @Test
    public void testMax() {
        System.out.println("max");
        int arrayNo = 0, initialValue = 1;
        int[] array;
        while (arrayNo < 1000) {
            array = constructArray(initialValue);
            assertEquals(initialValue + 9, array[ADiceGame.max(array)]);
            initialValue += 10;
            arrayNo++;
        }
    }

    /**
     * Test of cntOccurances method, of class ADiceGame.
     */
    @Test
    public void testCntOccurances() {
        System.out.println("cntOccurances");
        int[] array = {1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6,
6};
        assertEquals(1, ADiceGame.cntOccurances(array, 1));
        assertEquals(2, ADiceGame.cntOccurances(array, 2));
        assertEquals(3, ADiceGame.cntOccurances(array, 3));
        assertEquals(4, ADiceGame.cntOccurances(array, 4));
        assertEquals(5, ADiceGame.cntOccurances(array, 5));
        assertEquals(6, ADiceGame.cntOccurances(array, 6));
    }

    /**
     * Test of withdraw method, of class ADiceGame.
```

```
     */
    @Test
    public void testWithdraw() {
        System.out.println("withdraw");
        int[] score = {1100,5,1100};
        ADiceGame.withdraw(score);
        assertTrue(ADiceGame.withdraw(score));
        score[2] = 6;
        assertFalse(ADiceGame.withdraw(score));
    }

}
```

Κώδικας 8.9 *Test για την κλάση AdiceGame*

Καταρχάς να επισημάνουμε τον ρόλο μερικών νέων συναρτήσεων που περιλαμβάνονται στην ADiceGameTest και συναντούμε για πρώτη φορά.

assertEquals με δύο παραμέτρους ίδιου τύπου. Ελέγχει τις παραμέτρους και αν δεν είναι ίσες σημειώνει λάθος.

assertEquals με τρεις πραγματικές παραμέτρους. Ελέγχει τις δύο πρώτες παραμέτρους για προσεγγιστική ισότητα. Η τρίτη παράμετρος είναι η ανεκτή διαφορά. Στη δική μας περίπτωση χρησιμοποιούμε το EPSILON της εφαρμογής.

fail σημειώνει λάθος με μήνυμα την πραγματική παράμετρό της.

assertTrue σημειώνει λάθος εφόσον η παράμετρός της είναι ψευδής.

assertFalse σημειώνει λάθος εφόσον η παράμετρός της είναι αληθής.

Ας συζητήσουμε τώρα κάποιες από τις συναρτήσεις της ADiceGameTest ώστε να κατανοήσουμε πώς πρέπει να γράφουμε τις συναρτήσεις ελέγχου.

Η συνάρτηση testSum ελέγχει τη συνάρτηση sum της ADiceGame. Γνωρίζουμε ότι έχουμε τρεις πίνακες πραγματικών δηλωμένους στην ADiceGame, τον fair, τον lucky και τον unlucky και πως το άθροισμα των στοιχείων κάθε ενός από αυτούς τους πίνακες πρέπει να είναι ίσο με 1. Μπορούμε να αναπτύξουμε την testSum βασιζόμενοι σε αυτό το δεδομένο. Πράγματι, αν εξετάσετε τον κώδικα της testSum θα διαπιστώσετε περιλαμβάνει διαδοχικές κλήσεις στην sum, μία για κάθε έναν από τους fair, lucky και unlucky. Λαμβάνει το αποτέλεσμα της κλήσης στη μεταβλητή result την οποία συγκρίνει προσεγγιστικά μέσω της assertEquals με το 1. Αν η assertEquals εντοπίσει απόκλιση, θα σημειώσει λάθος.

Η testRoll καλεί την roll 10.000 φορές για κάθε μία κατανομή, δηλαδή για τους πίνακες fair, lucky και unlucky. Αν παραχθεί αριθμός έξω από το όριο 1 έως 6, καλεί την fail και σημειώνει το λάθος. Στο τέλος του βρόγχου καλείται η assertTrue για να ελέγξει τη συνθήκη sumLucky > sumFair && sumFair > sumUnlucky. Πράγματι, γνωρίζουμε πως η κατανομή lucky φέρνει περισσότερα άρτια αποτελέσματα από ότι η fair και η fair περισσότερα από ότι η unlucky. Μετά από 10.000 ρίψεις, οι πιθανότητες λογικά επαληθεύονται. Σε περίπτωση όμως που δεν επαληθευτούν η assertTrue θα σημειώσει λάθος.

Στην testMax δημιουργούμε 1000 πίνακες με τη βοήθεια της constructArray κατά τρόπο που γνωρίζουμε για κάθε πίνακα την τιμή του μέγιστου στοιχείου του. Προσέξτε πως η constructArray ανακατεύει τα στοιχεία του πίνακα που δημιουργεί έτσι ώστε το στοιχείο με τη μέγιστη τιμή να μην βρίσκεται πάντα στην τελευταία θέση του πίνακα. Στη συνέχεια ελέγχουμε αν η τιμή που επιστρέφει η max ανταποκρίνεται στην αναμενόμενη.

Εφόσον έχετε ακολουθήσει τα βήματα έως εδώ με ακρίβεια, το τεστ θα τρέξει επιτυχώς. Καλό είναι όμως να δείτε τι συμβαίνει σε περίπτωση που υπάρχει κάποια αποτυχία στο τεστ.
Μετατρέψτε προσωρινά τις cntOccurances και withdraw της ADiceGame όπως δείχνει ο κώδικας 8.10.

```
    public static int cntOccurances(int[] array, int value) {
        int cnt = 0;
        for (int i = 0; i < array.length; i++) {
            if (array[value] == array[i]) {
                cnt++;
            }
```

```
        }
        return cnt;
    }

    public static boolean withdraw(int[] score) {
        return cntOccurances(score, max(score)) > 1;
    }
```

Κώδικας 8.10 *Προσωρινή έκδοση της cntOccurances και withdraw.*

Μετά τη μετατροπή, τρέξτε και πάλι την κλάση ADiceGameTest. Μελετήστε το αποτέλεσμα. Επαναφέρετε τις cntOccurances και withdraw στην προηγούμενη σωστή μορφή τους.

8.9 Ασκήσεις προς λύση

8.9.1 Τεκμηρίωση, Απολαθοποίηση και Έλεγχος

Δίνεται η κλάση CharArrayLib στον κώδικα 8.11. Πρόκειται για μια βιβλιοθήκη διαχείρισης πινάκων χαρακτήρων. Στην main της κλάσης δημιουργούνται δύο πίνακες χαρακτήρων, ο s1={‘E’.’r’,’i’,’k’} και ο s2={‘s’,’o’,’n’}. Στη συνέχεια δημιουργείται ένας τρίτος πίνακας ως συνένωση των δύο πρώτων. Αμέσως μετά ελέγχεται αν ο πίνακας συνένωση περιλαμβάνει τον s1 και τον s2. Η έξοδος του κώδικα είναι 0 και -1. Αυτό σημαίνει πως ο πίνακας s1 βρέθηκε στον πίνακα συνένωση και αρχίζει στη θέση 0. Ο πίνακας s2 όμως δεν βρέθηκε ενώ περιέχεται στον πίνακα-συνένωση.
Να τεκμηριωθεί και να απολαθοποιηθεί η κλάση και να υλοποιηθεί κατάλληλος κώδικας ελέγχου.

```
public class CharArrayLib {

    static char[] create(String s) {
        return s.toCharArray();
    }

    static int compare(char[] s1, char[] s2) {
        int idx = 0;
        while (idx < s1.length && idx < s2.length) {
            if (s1[idx] > s2[idx]) {
                return 1;
            }
            if (s1[idx] < s2[idx]) {
                return -1;
            }
            idx++;
        }
        return s1.length-s2.length;
    }

    static int compareIgnoreCase(char[] s1, char[] s2) {
        int idx = 0;
        while (idx < s1.length && idx < s2.length) {
            char u1 = Character.toUpperCase(s1[idx]),
                 u2 = Character.toUpperCase(s2[idx]);
            if (u1 > u2) {
                return 1;
            }
            if (u1 < u2) {
                return -1;
            }
            idx++;
        }
```

```
        return s1.length - s2.length;
    }

    static char[] concat(char[] s1, char[] s2) {
        char[] rVal = new char[s1.length + s2.length];
        System.arraycopy(s1, 0, rVal, 0, s1.length);
        System.arraycopy(s2, 0, rVal, s1.length, s2.length);
        return rVal;
    }

    public static boolean equals(char[] tA, char[] tB) {
        if (tA.length != tB.length) {
            return false;
        }
        for (int i = 0; i < tA.length; i++) {
            if (tA[i] != tB[i]) {
                return false;
            }
        }
        return true;
    }

    static char[] subString(char[] s, int from, int to) {
        char[] rVal = new char[to - from];
        int idx = 0;
        for (int i = from; i < to; i++) {
            rVal[idx++] = s[i];
        }
        return rVal;
    }

    static int contains(char[] s, char... cS) {
        if (cS.length > s.length) {
            return -1;
        }

        for (int i = 0; i < s.length - cS.length; i++) {
            char[] sub = subString(s, i, cS.length + i);
            if (equals(sub, cS)) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        char[] s1 = create("Erik"), s2 = create("son");
        char[] con = concat(s1, s2);
        int idx = contains(con, s1);
        System.out.println(idx);
        idx = contains(con, s2);
        System.out.println(idx);
        char[] s4 =create("EriksoN");
        System.out.println(compareIgnoreCase(con,s4));
    }
}
```

Κώδικας 8.11 *Βιβλιοθήκη διαχείρισης πινάκων χαρακτήρων*

8.9.2 Παιχνίδι με τράπουλα

Τέσσερις φίλοι, ο John, ο Jim, η Mary και η Ann παίζουν χαρτιά. Το παιχνίδι που παίζουν έχει τους ακόλουθους κανόνες: Καταρχάς, ανακατεύεται η τράπουλα, 52 φύλλων (χωρίς μπαλαντέρ). Στον πρώτο γύρο τοποθετούνται τέσσερις κάρτες ανοικτές σε στοίβα στο τραπέζι και μοιράζονται από τέσσερις κάρτες σε κάθε παίκτη. Στους επόμενους γύρους λαμβάνουν από τέσσερις κάρτες μόνο οι παίκτες. Οι γύροι επαναλαμβάνονται έως ότου εξαντληθεί η τράπουλα.

Σε κάθε γύρο, οι παίκτες ανοίγουν τις κάρτες τους εναλλάξ μια προς μια και τις τοποθετούν στην κορυφή της στοίβας. Αν η κάρτα που άνοιξε ο παίκτης έχει ίδια τιμή με την κάρτα που βρίσκεται στην κορυφή της στοίβας, ο παίκτης κερδίζει τις κάρτες της στοίβας. Επίσης, κερδίζει τις κάρτες της στοίβας εφόσον η κάρτα που άνοιξε είναι βαλές. Όταν εξαντληθεί η τράπουλα, ο παίκτης που έχει τις περισσότερες κάρτες κερδίζει τις τυχόν εναπομείνασες κάρτες της στοίβας. Τέλος, οι παίκτες αθροίζουν τις τιμές των καρτών που έχουν μαζέψει και όποιος συγκεντρώνει το μεγαλύτερο άθροισμα, ανακηρύσσεται νικητής.

Να υλοποιηθεί προσομοίωση μιας παρτίδας παιχνιδιού μεταξύ των τεσσάρων φίλων. Να συνοδευτεί από κατάλληλο κώδικα ελέγχου.

Βιβλιογραφία

[1] L. B. Wilson and R. G. Clark, Comparative Programming Languages. Pearson Education, 2001.

[2] C. Böhm and G. Jacopini, “Flow diagrams, turing machines and languages with only two formation rules,” Commun. ACM, vol. 9, no. 5, pp. 366–371, May 1966, doi: 10.1145/355592.365646.

[3] “Exception handling,” Wikipedia. Sep. 19, 2021. Accessed: Oct. 15, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Exception_handling&oldid=1045188202

[4] K. Wiegers and J. Beatty, Software Requirements, 3rd edition. Redmond, Washington: Microsoft Press, 2013.

[5] “javadoc-The Java API Documentation Generator.” https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html (accessed Sep. 14, 2021).

[6] “How to Write Doc Comments for the Javadoc Tool.” https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html (accessed Sep. 14, 2021).

Κεφάλαιο 9

Σύνοψη
Σε αυτήν την ενότητα αναλύεται σε βάθος η έννοια της αναδρομής. Καταρχάς, αναλύεται η έννοια σε γενικότερο πλαίσιο, στη συνέχεια εξηγείται η λειτουργία της μνήμης stack (στοίβα) και μετά παρουσιάζονται οι προγραμματιστικές αναδρομικές τεχνικές, δηλαδή η άμεση και η αμοιβαία αναδρομή. Τέλος, περιλαμβάνεται συζήτηση για τα πλεονεκτήματα και μειονεκτήματα της αναδρομής σε σχέση με τις μη αναδρομικές προσεγγίσεις.

Προαπαιτούμενη γνώση
Οι θεμελιώδεις τύποι και βασική γνώση του τύπου String. Βασική είσοδος και έξοδος. Η λειτουργία των τελεστών. Οι δομές επιλογής και οι επαναληπτικές δομές. Η διαχείριση των πινάκων. Οι συναρτήσεις, η έννοια και βασική διαχείριση των απροσδόκητων λαθών, οι προσδιοριστές προσπέλασης και η χρήση των πακέτων.

Λέξεις κλειδιά
Μνήμη stack, αναδρομή, αμοιβαία αναδρομή

9 Αναδρομή

Όταν στον ορισμό μιας έννοιας χρησιμοποιείται η ίδια αυτή έννοια, λέμε πως έχουμε αναδρομικό ορισμό (recursive definition). Χαρακτηριστικό παράδειγμα αναδρομικού ορισμού είναι ο ορισμός της ακολουθίας Fibonacci που είδαμε στην άσκηση 7.8.3. Πιο συγκεκριμένα, αν Fn αναπαριστά τον ν-οστό όρο, η τιμή του υπολογίζεται ως εξής:

$$n = 0 \rightarrow F_n = 0$$
$$n = 1 \rightarrow F_n = 1$$
$$n > 1 \rightarrow F_n = F_{n-1} + F_{n-2}$$

Δηλαδή, αν το n είναι ίσο με το 0 ή το 1, ο ν-οστός όρος ισούται με το n. Για τιμές μεγαλύτερες του 1, ο ν-οστός όρος ισούται με το άθροισμα των δύο προηγούμενων όρων. Επομένως, η τιμή ενός όρου της ακολουθίας Fibonacci ορίζεται με βάση την τιμή δύο όρων της ίδιας ακολουθίας.

Πώς όμως μπορούμε να υπολογίσουμε το F3; Με βάση αυτόν τον ορισμό, F3=F2+F1. Το F1 έχει τιμή 1. Δεν γνωρίζουμε όμως την τιμή του F2. Ωστόσο, γνωρίζουμε πως F2=F1+F0. Επομένως, F2=1 και άρα F3=2.

Ένα άλλο χαρακτηριστικό παράδειγμα είναι ο αναδρομικός ορισμός της δύναμης ενός αριθμού υψωμένου σε εκθέτη. Ας πάρουμε την απλή περίπτωση που η βάση είναι πραγματικός και ο εκθέτης ακέραιος. Σε αυτήν την περίπτωση ο γνωστός μας μη αναδρομικός ορισμός έχει ως εξής;

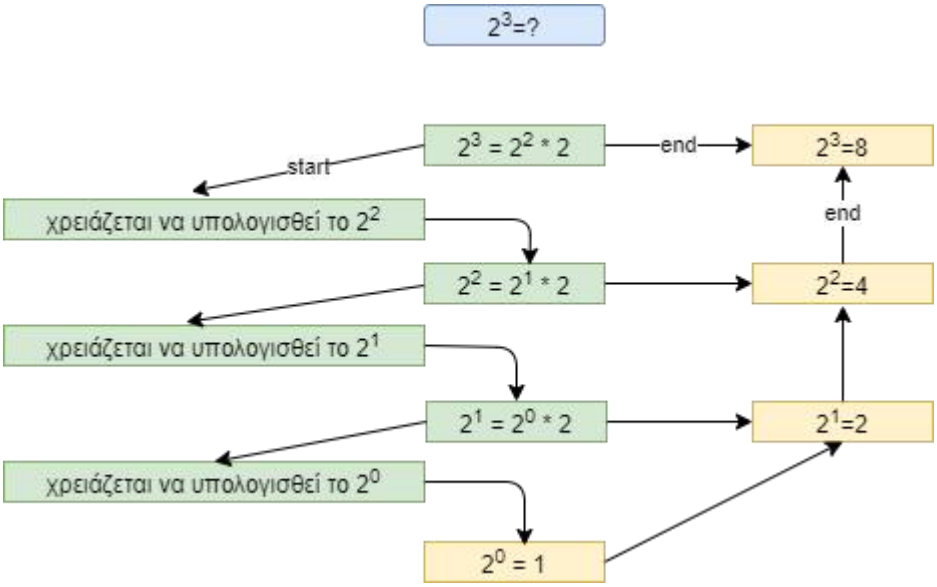
$$a^e = \begin{cases} 1 \ \alpha\nu \ e = 0 \\ \alpha * a * \ldots a, \ e \ \varphi o\rho\acute{\varepsilon}\varsigma \ \alpha\nu \ e > 0 \\ \dfrac{1}{a * a * \ldots a, \ e \ \varphi o\rho\acute{\varepsilon}\varsigma} \ \alpha\nu \ e < 0 \end{cases}$$ [8]

Όμως ισχύει και ο αναδρομικός ορισμός.

[8] Ας σημειωθεί εδώ πως το 0^0 δεν ορίζεται. Ωστόσο σε πολλές περιπτώσεις θεωρείται πως 0^0=1. Πολλές γλώσσες προγραμματισμού συμπεριλαμβανομένων των Java, Javascript και C, C++, υπολογίζουν το 0^0 ίσο με 1. Για παράδειγμα, αν καλέσετε τη συνάρτηση Math.pow(0, 0), θα διαπιστώσετε πως επιστρέφει 1. Για συμβατότητα θεωρούμε σε αυτό το βιβλίο πως το $0^0 = 1$.

$$a^e = \begin{cases} 1, \alpha\nu\ e = 0 \\ a^{e-1} * a, \alpha\nu\ e > 0 \\ \frac{1}{a^{-e}}, \alpha\nu\ e < 0 \end{cases}$$

Όπως βλέπουμε στον αναδρομικό ορισμό της δύναμης το a^e ορίζεται ως το $a^{e-1}*a$ όταν το e είναι θετικός και ως $1/a^{-e}$ όταν το e είναι αρνητικός. Πώς θα υπολογίσουμε αναδρομικά την τιμή του 2^3; Ας το συζητήσουμε με τη βοήθεια του σχήματος 9.1.



Σχήμα 9.1 *Ο αναδρομικός υπολογισμός του 2^3*

Όπως φαίνεται στο σχήμα 9.1, για να υπολογίσουμε το 2^3 θα πρέπει πρώτα να υπολογίσουμε το 2^2. Παρόμοια, για να υπολογίσουμε το 2^2, θα πρέπει πρώτα να υπολογίσουμε το 2^1 και για να υπολογίσουμε το 2^1 θα πρέπει να υπολογίσουμε το 2^0. Το 2^0 όμως εξ ορισμού είναι ίσο με το 1. Επομένως τώρα εύκολα υπολογίζουμε το 2^1 που είναι ίσο με 2. Γνωρίζοντας το 2^1, μπορούμε να υπολογίσουμε το 2^2 που είναι ίσο με 4. Τέλος, γνωρίζοντας το 2^2, εύκολα υπολογίζουμε το 2^3 που είναι ίσο με 8.

Γενικότερα, η αναδρομή αποδομεί ένα πρόβλημα τάξης ν σε ένα ή περισσότερα ίδια προβλήματα τάξης μικρότερης του ν. Η διαδικασία της αποδόμησης επαναλαμβάνεται έως ότου φτάσουμε σε ένα ή περισσότερα απλά προβλήματα με προφανή λύση. Τότε τα αποτελέσματα από τα προφανή προβλήματα αξιοποιούνται για τη λύση των προβλημάτων του αμέσως ανώτερου επιπέδου των οποίων η λύση επίσης συνεισφέρει στη λύση των προβλημάτων του ανώτερου επιπέδου, κοκ, έως ότου φθάσουμε στη λύση του αρχικού προβλήματος.

Στη συνέχεια, θα δούμε την αναδρομή στην Java και θα συζητήσουμε τα πλεονεκτήματα και μειονεκτήματά της. Πρώτα όμως θα δώσουμε μερικές απαραίτητες πληροφορίες για μια περιοχή της μνήμης ενός προγράμματος που ονομάζεται στοίβα (stack).

9.1 Η λειτουργία της στοίβας

Για ευκολία θα υποθέσουμε πως κάθε εφαρμογή Java διαθέτει μια μνήμη που ονομάζεται στοίβα[9]. Σύμφωνα με τις προδιαγραφές (specification) της Java [1], η στοίβα μπορεί να είναι σταθερού ή δυναμικά μεταβαλλόμενου μεγέθους. Και στις δύο περιπτώσεις, η στοίβα μπορεί να γεμίσει με δεδομένα. Στην πρώτη περίπτωση, αφού καλυφτεί το προκαθορισμένο μέγεθός της και στη δεύτερη αφού καταλάβει όλη τη διαθέσιμη δυναμική μνήμη. Αν το πρόγραμμά μας απαιτεί επιπλέον μνήμη στοίβας, τότε έχουμε αδυναμία συνέχισης της εκτέλεσης του προγράμματος. Επομένως, η εξάντληση της στοίβας είναι ένα πολύ σοβαρό πρόβλημα που πάντα πρέπει να λαμβάνει ο προγραμματιστής υπόψη του.

---

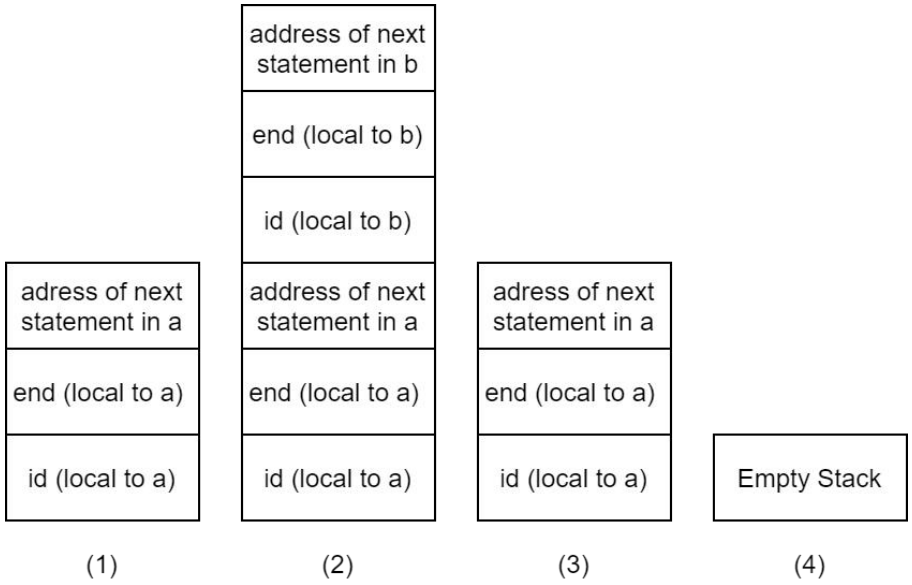
[9] Στην πραγματικότητα, στις πολυνηματικές (multithreading) εφαρμογές, διαμορφώνεται μια στοίβα για κάθε νήμα.

Ας δούμε όμως τι αποθηκεύεται στη στοίβα και ποια μπορεί να είναι η πιθανή αιτία εξάντλησής της. Στον κώδικα 9.2 παρουσιάζουμε μια απλή μελέτη περίπτωσης.

```
public class StackFunctionality {

    static void a() {
        String id = "static function a";
        System.out.println("start " + id);
        String end = "end";
        b();
        System.out.println(end + " " + id);
    }

    static void b() {
        String id = "static function b";
        System.out.println("start " + id);
        String end = "end";
        c();
        System.out.println(end + " " + id);
    }

    static void c() {
        String id = "static function c";
        System.out.println("start " + id);
        String end = "end";
        System.out.println("c is running");
        System.out.println(end + " " + id);
    }

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

Κώδικας 9.1 *Περίπτωση μελέτης λειτουργίας της στοίβας*

Όπως βλέπουμε στον κώδικα 9.1, η συνάρτηση a καλεί την b και αυτή με τη σειρά της, την c. Όταν όμως η c θα ολοκληρωθεί, θα πρέπει η b να συνεχίσει. Από ποιο σημείο όμως; Μα από τη γραμμή της a που έπεται της κλήσης της c. Το ίδιο θα συμβεί όταν ολοκληρωθεί η εκτέλεση της b, δηλαδή θα πρέπει να συνεχίσει η a. Προφανώς, από τη γραμμή του κώδικα που έπεται της κλήσης της b. Η συνάρτηση a προσπελαύνει τις μεταβλητές end και id. Όμως, η b διαθέτει δικές της τέτοιες μεταβλητές. Η προσπέλαση με τα αναγνωριστικά end και id μέσα στην b αναφέρεται στις τοπικές μεταβλητές της b. Αντίθετα, οι τοπικές μεταβλητές της a είναι απροσπέλαστες κατά την εκτέλεση της b. Επομένως, είναι φανερό πως η κλήση συνάρτησης μέσα από άλλη συνάρτηση απαιτεί μια ειδική διαχείριση. Η διαχείριση αυτή βασίζεται στη στοίβα.
Η στοίβα είναι μια Last In First Out (LIFO) δομή, δηλαδή ό,τι μπαίνει τελευταίο στη στοίβα βγαίνει πρώτο ή αλλιώς ό,τι εισάγεται στη στοίβα, τοποθετείται στην κορυφή της και ό,τι εξάγεται, λαμβάνεται από την κορυφή της.



Σχήμα 9.2 *Η στοίβα κατά την εκτέλεση της main του κώδικα 9.1*

Ας δούμε όμως πώς λειτουργεί η στοίβα κατά την εκτέλεση του κώδικα 9.1. Για απλοποίηση, στην περιγραφή αυτή παραλείπουμε τις κλήσεις πέραν των κλήσεων της a, της b και της c. Καταρχάς, έχουμε την κλήση της a. Η a κατανέμει τις τοπικές μεταβλητές της και προχωρά στην εκτέλεσή της μέχρι του σημείου που καλεί την b. Πριν ακριβώς ξεκινήσει η b, οι τοπικές μεταβλητές της a αποθηκεύονται στη στοίβα όπως δείχνει το σχήμα 9.2-1. Όπως φαίνεται σε αυτό το σχήμα, εκτός από τις τοπικές μεταβλητές, στη στοίβα αποθηκεύεται και η διεύθυνση του κώδικα από την οποία θα συνεχιστεί η εκτέλεση της a όταν η b θα έχει τερματίσει. Στη συνέχεια εκτελείται η b μέχρι του σημείου που καλείται η c. Πριν την κλήση της c, οι τοπικές μεταβλητές της b και η διεύθυνση της γραμμής που έπεται της κλήσης της c, επίσης αποθηκεύονται στη στοίβα. Έτσι, η στοίβα διαμορφώνεται όπως δείχνει το σχήμα 9.2-2. Με τον τερματισμό της c, η b εξάγει τις απαραίτητες πληροφορίες από την κορυφή της στοίβας και συνεχίζει την εκτέλεσή της, οπότε η στοίβα διαμορφώνεται όπως δείχνει το σχήμα 9.2-3. Παρόμοια, με τον τερματισμό της b, η a εξάγει από τη στοίβα τις πληροφορίες που την αφορούν και οι οποίες μετά την εξαγωγή των πληροφοριών της b, βρίσκονται στην κορυφή της στοίβας. Έτσι, η a συνεχίζει απρόσκοπτα τη λειτουργία της, ενώ η στοίβα έχει αδειάσει (σχήμα 9.2-4).

Επομένως, κάθε φορά που μια συνάρτηση καλείται από μία άλλη, στη στοίβα τοποθετούνται διάφορες πληροφορίες. Άρα είναι δυνατόν αν έχουμε πολύ μεγάλο βάθος κλήσεων, η στοίβα να μην διαθέτει άλλο χώρο. Ωστόσο, κάτι τέτοιο είναι απίθανο να συμβεί μέσα από μη αναδρομικές κλήσεις. Μπορεί όμως να συμβεί από αναδρομικές κλήσεις. Πόσες φορές θα καλέσει μία συνάρτηση τον εαυτό της, εξαρτάται από το μέγεθος του προβλήματος που επιχειρεί να επιλύσει.

9.2 Αναδρομικές συναρτήσεις

Σύμφωνα με τα όσα έχουμε πει, είναι εύκολο να υποθέσουμε πως μια συνάρτηση είναι αναδρομική όταν καλεί τον εαυτό της. Προσέξτε τη συνάρτηση endless στον κώδικα 9.2.

```
static void endless(int i) { //Κώδικας 9.2
    System.out.println(i);
    endless(i + 1);
}
```

Κώδικας 9.2 *Αναδρομική συνάρτηση χωρίς συνθήκη ελέγχου*

Καταρχάς, είναι φανερό πως πρόκειται για αναδρομική συνάρτηση. Αν καλέσουμε endless(0), τότε η συνάρτηση θα εμφανίσει 0 και στη συνέχεια θα καλέσει endless(1). Η endless(1) θα εμφανίσει 1 και θα καλέσει endless(2). Παρόμοια, η endless(2) θα εμφανίσει 2 και θα καλέσει endless(3). Η αναδρομική κλήση θα συνεχιστεί στην ουσία μέχρις ότου γεμίσει η στοίβα και το πρόγραμμα πέσει.

Επομένως, κάθε αναδρομική συνάρτηση πρέπει να περιλαμβάνει μια συνθήκη ελέγχου της αναδρομικής κλήσης. Με άλλα λόγια, μια αναδρομική συνάρτηση πρέπει να περιλαμβάνει κάποιο κώδικα που κάτω από κάποιες συνθήκες θα τερματίζει την αναδρομική κλήση.

```
static void withEnd(int i) { //Κώδικας 9.3
    System.out.println(i);
    if (i < 3) {
        withEnd(i + 1);
    }
}
```

Κώδικας 9.3 *Απλή αναδρομή με συνθήκη τερματισμού*

Όπως φαίνεται στον κώδικα 9.3, η withEnd είναι επίσης αναδρομική. Ωστόσο, η αναδρομική κλήση γίνεται μόνο εφόσον το i είναι μικρότερο του 3. Για παράδειγμα, αν καλέσουμε withEnd(0), η συνάρτηση θα εμφανίσει 0, στη συνέχεια ελέγχει το i, το βρίσκει μικρότερο του 3 καλεί την endless(1). Με την ίδια διαδικασία, τυπώνει 1, μετά 2, καλεί και withEnd(3), εμφανίζει και το 3 και μετά βρίσκει ότι η έκφραση i<3 είναι ψευδής και σταματά την αναδρομική κλήση. Σε αυτήν την περίπτωση μπορούμε να πούμε ότι η withEnd δεν πέφτει. Ισχύει όμως αυτό γενικευμένα; Η απάντηση είναι όχι. Όπως αναφέραμε και προηγουμένως, εξαρτάται από το μέγεθος του προβλήματος. Για παράδειγμα, αν καλέσουμε withEnd(Integer.MIN_VALUE), θα έχουμε τόσες αναδρομικές κλήσεις ώστε η συνάρτηση θα πέσει πολύ πριν κληθεί με παράμετρο 3.

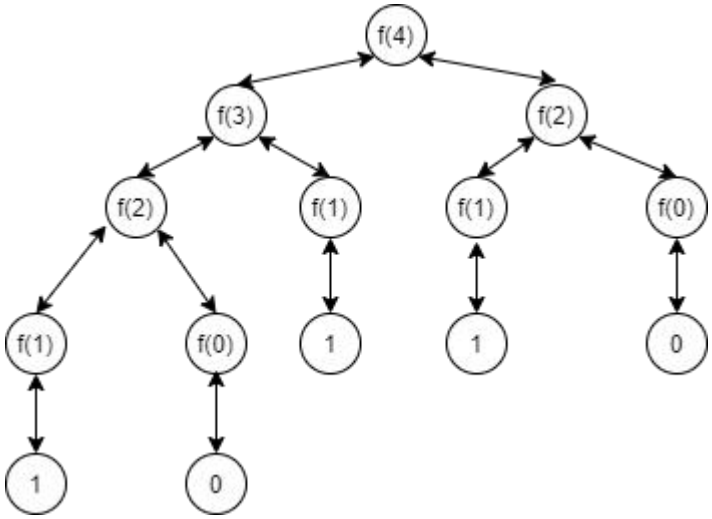
Αυτό είναι και το βασικό μειονέκτημα της αναδρομής. Οποιαδήποτε αναδρομική συνάρτηση μπορεί να αντιμετωπίσει πρόβλημα χωρητικότητας της στοίβας ανάλογα με το μέγεθος του προβλήματος που επιχειρεί να αντιμετωπίσει.

Στον κώδικα 9.4 παραθέτουμε την αναδρομική υλοποίηση υπολογισμού του ν-οστού όρου της ακολουθίας Fibonacci.

```
static int fibonacci(int n) { //Κώδικας 9.4
    if (n < 0) {
        throw new RuntimeException();
    }
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Κώδικας 9.4 *Αναδρομική υλοποίηση υπολογισμού το ν-οστού όρου της ακολουθίας Fibonacci*

Στην ακολουθία Fibonacci δεν ορίζονται όροι αρνητικής τάξης. Έτσι αν η συνάρτηση κληθεί με αρνητικό όρισμα παράγει εξαίρεση. Στη συνέχεια, ο κώδικας 9.4 υλοποιεί ακριβώς τον ορισμό της ακολουθίας που δόθηκε στην αρχή της ενότητας 9. Ας δούμε πώς λειτουργεί η συνάρτηση όταν κληθεί με όρισμα 4. Στο σχήμα 9.3, για απλοποίηση, η συνάρτηση συμβολίζεται με f αντί για fibonacci.



Σχήμα 9.3 *Κλήσεις και τιμές επιστροφής της fibonacci*

Όπως βλέπουμε στο σχήμα 9.3, καταρχάς έχουμε κλήση της f(4). Σύμφωνα με τον κώδικα 9.4, η κλήση αυτή θα επιστρέψει f(3)+f(2). Επομένως, καλείται η f(3) και η f(2) όπως δείχνει και το σχήμα 9.3. Στη συνέχεια, ο υπολογισμός της f(3) απαιτεί την κλήση της f(2) και f(1) και ο υπολογισμός της f(2) απαιτεί την κλήση f(1) και f(0). Σε αυτό το βάθος τρεις κόμβοι του δέντρου, οι f(1), f(1) και f(0), υπολογίζονται χωρίς περαιτέρω αναδρομική κλήση. Αντίθετα ο κόμβος f(2) απαιτεί την κλήση των f(1) και f(0). Κάθε τερματικός κόμβος επιστρέφει την τιμή του στο αμέσως προηγούμενο επίπεδο. Έτσι το f(1) υπολογίζεται ως 1, το f(0) ως 0, το f(2) ως 1, το f(3) ως 2 και το f(4) ως 3.

Ας σημειωθεί πως η περίπτωση κατά την οποία το n είναι μικρότερο ή ίσο του 1 ονομάζεται βασική περίπτωση (base case). Γενικότερα, στην αναδρομή, βασική περίπτωση ονομάζεται η περίπτωση κατά την οποία υπάρχει προφανής λύση που ο υπολογισμός της δεν απαιτεί αναδρομική κλήση.

Το παράδειγμα που θα δούμε τώρα είναι απλό, ωστόσο ιδιαίτερα κρίσιμο για την κατανόηση του μηχανισμού αναδρομικών κλήσεων.

```
static void prt(int i) { // Κώδικας 9.5 prt(1) 123321
    System.out.print(i);
    if (i < 3) {
        prt(i + 1);
    }
    System.out.print(i);
}
```

Κώδικας 9.5 *Συνέχεια της συνάρτησης μετά την αναδρομική κλήση*

Προσέξτε τη συνάρτηση prt του κώδικα 9.5. Τι εμφανίζει κατά την άποψή σας η κλήση prt(3); Συχνά, λαμβάνω λανθασμένη απάντηση σε αυτήν την ερώτηση. Πολλοί φοιτητές στο στάδιο της εκμάθησης της αναδρομής θεωρούν πως η prt(0) θα εμφανίσει 0123. Για να το δούμε βήμα προς βήμα. Όταν αρχίσει η εκτέλεση της prt(0), είναι προφανές πως καταρχάς θα εμφανίσει 0. Στη συνέχεια, ελέγχει τη συνθήκη i<3 την οποία βρίσκει αληθή και εκτελεί το μπλοκ της if. Με την ολοκλήρωση του μπλοκ της if, ο έλεγχος επανέρχεται στην κλήση prt(0). Η συνάρτηση εξάγει την τιμή της i από τη στοίβα και προχωράει στην εκτέλεση της τελευταίας εντολής της στην οποία εμφανίζει και πάλι το 0. Άρα, μέχρι εδώ έχουμε υπολογίσει μια έξοδο ως εξής: 0[Έξοδος από το μπλοκ της if, με i==0]0. Στο μπλοκ της if καλείται καταρχάς, η prt(1) η οποία λειτουργεί ακριβώς με τον ίδιο τρόπο, δηλαδή το αποτέλεσμά της μπορεί να περιγραφεί ως 1[Έξοδος από το μπλοκ της if, με i==1]1 ενώ το συνολικό αποτέλεσμα μέχρι στιγμής είναι 01[Έξοδος από το μπλοκ της if, με i==1]01. Συνεχίζοντας με αυτόν τον τρόπο φτάνουμε στο τελικό αποτέλεσμα που είναι 01233210.

Το κλειδί εδώ είναι να προσέξουμε το εξής: Η prt(0) καλεί κάποια στιγμή την prt(1). Αυτό όμως δεν σημαίνει πως η prt(0) έχει ολοκληρωθεί. Όταν η αναδρομική κλήση θα ολοκληρωθεί, η prt(0) θα συνεχίσει από την επόμενη γραμμή κώδικα από όπου έγινε η κλήση prt(1).

Ας δούμε τώρα την αναδρομική συνάρτηση δύναμης με βάση πραγματικό και εκθέτη ακέραιο.

```
static double power(double b, int e) { //Κώδικας 9.6
    if (e > 0) {
        return power(b, e - 1) * b;
    }
    if (e < 0) {
        return 1 / power(b, -e);
    }
    return 1;
}
```

Κώδικας 9.6 *Αναδρομική υλοποίηση του υπολογισμού δύναμης*

Όταν έχουμε διαθέσιμο έναν αναδρομικό ορισμό, είναι εύκολο να υλοποιήσουμε την αντίστοιχη αναδρομική συνάρτηση. Θυμηθείτε τον αναδρομικό ορισμό της δύναμης στην αρχή της ενότητας 9 και συγκρίνετέ τον με τον κώδικα 9.6. Θα διαπιστώσετε σημαντική ομοιότητα. Στην ουσία, πρόκειται για απλή μεταγραφή από τη γλώσσα των μαθηματικών στη γλώσσα του προγραμματισμού.

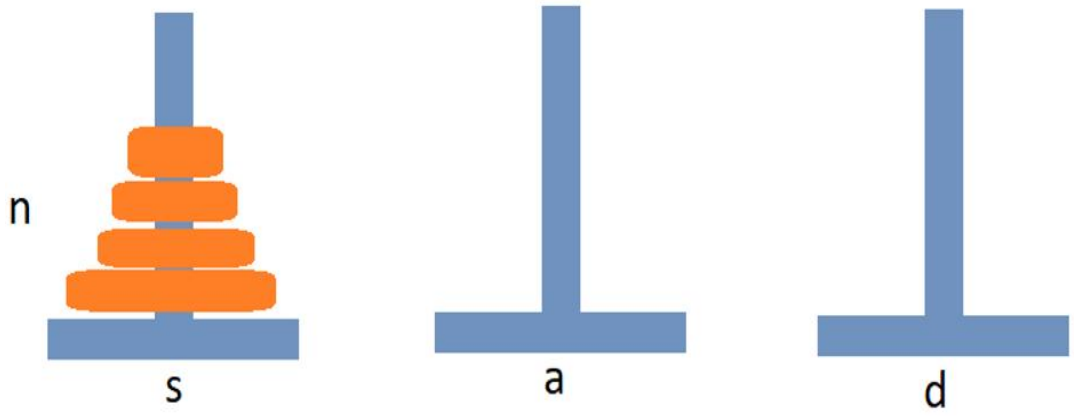
Η συνάρτηση power ελέγχει τον εκθέτη e, αν είναι θετικός επιστρέφει power(b, e-1)*b, δηλαδή ακριβώς ό,τι λέει και ο μαθηματικός ορισμός. Αν πάλι ο εκθέτης είναι αρνητικός, επιστρέφει 1/power(b,-e). Τέλος, εφόσον ο εκθέτης είναι 0, δεν θα γίνουν οι αναδρομικές κλήσεις που προβλέπονται για θετικό και αρνητικό εκθέτη, οπότε η συνάρτηση θα επιστρέψει 1.

9.3 Οι πύργοι του Ανόι

Οι πύργοι του Ανόι (towers of Hanoi) είναι ένα μαθηματικό πρόβλημα που χρησιμοποιείται ευρύτατα για την εκμάθηση της αναδρομής στον προγραμματισμό. Η λύση του φαίνεται αρχικά αρκετά περίπλοκη σε σχέση με τις λύσεις των προβλημάτων που έχουμε δει ως τώρα. Ωστόσο, αν σκεφτεί κανείς με τον κατάλληλο τρόπο, θα διαπιστώσει πως η αναδρομική λύση απαιτεί περιορισμένο κώδικα και τελικά αποδεικνύεται απλούστερη της μη αναδρομικής.

9.3.1 Περιγραφή του προβλήματος

Έχουμε τρεις στύλους, τον s από το source (προέλευση), τον d από το destination (προορισμός) και τον a από το auxiliary (βοηθητικός). Στον στύλο s είναι τοποθετημένοι n δίσκοι όπως φαίνεται στο σχήμα 9.4. Πιο συγκεκριμένα, οι δίσκοι έχουν διαφορετική διάμετρο και είναι τοποθετημένοι έτσι ώστε ένας δίσκος με μεγαλύτερη διάμετρο δεν βρίσκεται ποτέ επάνω από έναν δίσκο με μικρότερη διάμετρο.



Σχήμα 9.4 *Οι πύργοι του Ανόι*

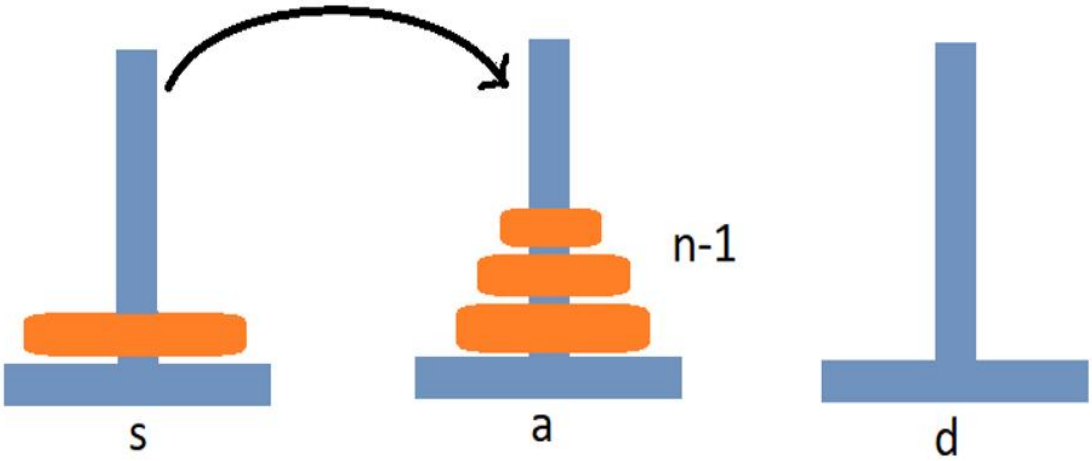
Το ζητούμενο είναι να μεταφέρουμε τους δίσκους από τον στύλο s στον στύλο d χρησιμοποιώντας ως βοηθητικό τον στύλο a. Η μετακίνηση όμως θα πρέπει να γίνει με τους εξής κανόνες:

- Κάθε φορά μπορούμε να μετακινήσουμε έναν μόνο δίσκο από έναν στύλο σε έναν άλλο.

- Δεν πρέπει ποτέ ένας δίσκος με μεγαλύτερη διάμετρο να τοποθετηθεί επάνω από έναν δίσκο με μικρότερη διάμετρο.

9.3.2 Λύση

Η απλούστερη οδός για να λύσουμε το πρόβλημα με αναδρομή είναι να σκεφτούμε με τη φιλοσοφία της αναδρομής. Ας υποθέσουμε πως έχουμε κάποιο τρόπο για να μεταφέρουμε τους n-1 δίσκους από το s στο a χρησιμοποιώντας ως βοηθητικό το d. Σε αυτήν την περίπτωση, στο s θα έχει μείνει ένας δίσκος και στο a θα έχουν τοποθετηθεί οι n-1 δίσκοι όπως δείχνει το σχήμα 9.5. Επομένως, ο δίσκος στο s μπορεί να μετακινηθεί στο d χωρίς πρόβλημα. Αν υποθέσουμε τώρα πως με τον ίδιο τρόπο που μεταφέραμε τους n-1 στύλους στο a, θα τους μεταφέρουμε στο d με βοηθητικό το s, τότε έχουμε λύσει το πρόβλημα. Επομένως, η μεταφορά n δίσκων από το s στο d με βοηθητικό το a έχει αναλυθεί σε δύο προβλήματα μεταφοράς n-1 δίσκων. Αν λύσουμε τα δύο αυτά προβλήματα, έχουμε λύση και στο αρχικό μας πρόβλημα.



Σχήμα 9.5 *Οι n-1 δίσκοι έχουν μετακινηθεί στον στύλο a χρησιμοποιώντας ως βοηθητικό τον στύλο d.*

Πώς θα λύσουμε τώρα αυτά τα δύο προβλήματα; Με τον ίδιο τρόπο. Στο πρώτο πρόβλημα, έχουμε να μεταφέρουμε n-1 δίσκους από το s στο a χρησιμοποιώντας ως βοηθητικό το d. Αν μεταφέρουμε τους n-2 δίσκους από το s στο d χρησιμοποιώντας ως βοηθητικό το a, στον s θα έχουν μείνει δύο στύλοι και ο a θα είναι άδειος, οπότε ο στύλος στην κορυφή του s μπορεί να μετακινηθεί χωρίς πρόβλημα στον a. Στη συνέχεια θα πρέπει να μεταφέρουμε τους n-2 από το d στο a. Επομένως έχουμε αναλύσει το ένα πρόβλημα μεταφοράς n-1 δίσκων σε δύο προβλήματα μεταφοράς n-2 δίσκων.

Η ανάλυση μπορεί να συνεχιστεί για n-1 βήματα. Στο πρώτο βήμα αναλύεται το πρόβλημα μεγέθους n σε δύο υποπροβλήματα μεγέθους n-1, στο δεύτερο βήμα αναλύονται τα δύο υποπροβλήματα μεγέθους n-1 σε τέσσερα υποπροβλήματα μεγέθους n-2. Η διαδικασία συνεχίζεται και στο βήμα n-1 έχει αναλυθεί το πρόβλημα σε 2n-1 υποπροβλήματα μεγέθους n-(n-1) το καθένα, δηλαδή υποπροβλήματα μεγέθους 1. Κάθε πρόβλημα όμως μεγέθους 1, έχει προφανή λύση όπως έχουμε ήδη εξηγήσει. Επομένως, η λύση σε αυτό το επίπεδο, δίνει τη δυνατότητα λύσης στο αμέσως προηγούμενο το οποίο με τη σειρά του δίνει λύση στο προηγούμενό του επίπεδο. Μέσα από αυτήν την αναδρομική διαδικασία φτάνουμε και στη λύση του αρχικού προβλήματος.

```
public class TowersOfAnoi {

    private static void moveOne(int diskNo, char from, char to) {
            System.out.println("move disk " + diskNo + " from " + from + " to " +
to);
    }

    public static void moveDisks(int n, char s, char d, char a) {
        if (n == 1) {
            moveOne(n, s, d);
```

```
        } else {
            moveDisks(n - 1, s, a, d);
            moveOne(n, s, d);
            moveDisks(n - 1, a, d, s);
        }
    }

    public static void main(String[] args) {
        moveDisks(4,'S','D','A');
    }
}
```

Κώδικας 9.7 *Αναδρομική λύση του προβλήματος towers of Hanoi*

Όπως δείχνει ο κώδικας 9.7, έχουμε κωδικοποιήσει το πρόβλημα ως τη συνάρτηση moveDisks που λαμβάνει τέσσερις παραμέτρους. Η παράμετρος n αντιπροσωπεύει τον αριθμό των δίσκων και οι υπόλοιπες τρεις παράμετροι τους ομώνυμους στύλους s, d και a. Με αυτήν την αναπαράσταση μπορούμε να θεωρήσουμε ότι κάθε δίσκος έχει μία ταυτότητα. Ο δίσκος που βρίσκεται πάνω-πάνω στη στοίβα των δίσκων έχει την ταυτότητα 1, ο αμέσως επόμενος την ταυτότητα 2. Έτσι χαρακτηρίζονται όλοι οι δίσκοι μέχρι τον δίσκο στη βάση της στοίβας που είναι ο δίσκος n.

Παράλληλα, έχουμε και τη συνάρτηση moveOne. Πρόκειται για μια απλή συνάρτηση που το μόνο που κάνει είναι να εμφανίζει ένα μήνυμα που μας πληροφορεί ποιος δίσκος μετακινήθηκε και από ποιον στύλο έφυγε και σε ποιον τοποθετήθηκε.

Κατά τα λοιπά, η moveDisks εφαρμόζει ακριβώς τη λογική που περιγράψαμε. Πιο συγκεκριμένα, αν κληθεί με n=1, τότε η μετακίνηση του δίσκου είναι μια απλή μετακίνηση, οπότε καλεί την moveOne. Διαφορετικά, μεταφέρει τους n-1 δίσκους από το s στο a με βοηθητικό το d. Μετά μεταφέρει τον μοναδικό δίσκο στο s και τον τοποθετεί στο d και τέλος μεταφέρει τους n-1 από το a όπου τους έχει τοποθετήσει προηγουμένως στο d χρησιμοποιώντας ως βοηθητικό το s.

9.4 Αμοιβαία αναδρομή

Η αναδρομή κατά την οποία μια συνάρτηση καλεί τον εαυτό της ονομάζεται ειδικότερα άμεση αναδρομή (direct recursion). Είναι όμως πιθανό μια συνάρτηση a να καλεί μια συνάρτηση b και η b να καλεί την a. Πρόκειται και πάλι για αναδρομή που όμως ονομάζεται αμοιβαία αναδρομή (mutual) ή έμμεση (indirect) αναδρομή.

Ας δούμε ένα πρόβλημα που μπορεί να επιλυθεί με αμοιβαία αναδρομή. Έστω ότι έχουμε μια διαδικασία εξέτασης και μια διαδικασία αξιολόγησης της εξέτασης. Η εξέταση βαθμολογεί ένα γραπτό και καλεί την αξιολόγηση. Η δε αξιολόγηση ελέγχει τον βαθμό και αν είναι προβιβάσιμος ενημερώνει με κατάλληλο μήνυμα. Αν όχι, πάλι ενημερώνει με κατάλληλο μήνυμα αλλά καλεί επαναληπτική εξέταση. Η διαδικασία συνεχίζεται μέχρις ότου επιτευχθεί προβιβάσιμος βαθμός.

Ο κώδικας 9.8 δίνει λύση σε αυτό το πρόβλημα με αμοιβαία αναδρομή;

```
    static void exam() { //Κώδικας 9.8
        Random generator = new Random();
        int mark = generator.nextInt(10) + 1;
        evaluate(mark);
    }

    static void evaluate(int mark) { //Κώδικας 9.8
        System.out.print("Your mark is: " + mark);
        if (mark >= 7) {
            System.out.println(" Success");
        } else {
            System.out.println(" Fail");
            exam();
        }
    }
```

Κώδικας 9.8 *Εξέταση και αξιολόγηση με αμοιβαία αναδρομή*

Η συνάρτηση exam παράγει έναν τυχαίο ακέραιο από 1 έως 10. Ο αριθμός αυτός υποτίθεται πως αναπαριστά τον βαθμό ενός γραπτού. Στη συνέχεια καλεί τη συνάρτηση evaluate με παράμετρο τον βαθμό. Η evaluate με τη σειρά της ανακοινώνει τον βαθμό και ανάλογα με την τιμή του ενημερώνει πως έχουμε επιτυχία ή αποτυχία. Στην περίπτωση της αποτυχίας καλεί και πάλι την exam.
Παρόμοια με τη συνάρτηση fibonacci που ορίζεται αναδρομικά, οι συναρτήσεις Hofstadter male και female [2] ορίζονται με αμοιβαία αναδρομή. Πρόκειται για δύο σειρές ακέραιων που δίνονται από τον ακόλουθο ορισμό:

$$f(0) = 1$$
$$m(0) = 0$$
$$f(n) = n - m\big(f(n-1)\big),\ n > 0$$
$$m(n) = n - f\big(m(n-1)\big),\ n > 0$$

Ο κώδικας 9.9 δίνει την υλοποίηση σε Java των συναρτήσεων f και m.

```
static int f(int n) {
    if (n == 0) {
        return 1;
    }
    return n - m(f(n - 1));
}

static int m(int n) {
    if (n == 0) {
        return 0;
    }
    return n - f(m(n - 1));
}
```

Κώδικας 9.9 *Υπολογισμός σειρών Hofstadter male και female*

Παρότι η υλοποίηση των συναρτήσεων με δεδομένο τον αμοιβαία αναδρομικό ορισμό τους είναι σχετικά εύκολη, η παρακολούθηση των κλήσεων και των επιστροφών είναι αρκετά περίπλοκη. Στον πίνακα 9.1, παραθέτουμε τις κλήσεις και επιστροφές που προκαλούνται προκειμένου να υπολογιστούν οι m(2) και f(2).

1	m(2)	f(2)
2	m(1)	f(1)
3	m(0)	f(0)
4	m(0)->0	f(0)->1
5	f(0)	m(1)
6	f(0)->1	m(0)
7	m(1)->0	m(0)->0
8	f(0)	f(0)
9	f(0)->1	f(0)->1
10	m(2)->1	m(1)->0
		f(1)->1
		m(1)
		m(0)
		m(0)->0
		f(0)
		f(0)->1
		m(1)->0
		f(2)->2

Πίνακας 9.2 *Κλήσεις και επιστροφές για τον υπολογισμό των m(2) και f(2)*

Στον πίνακα 9.2 ένα στοιχείο που δεν περιέχει τον χαρακτήρα -> επισημαίνει την είσοδο στην αντίστοιχη συνάρτηση. Αντίθετα τα στοιχεία που περιέχουν -> επισημαίνουν την επιστροφή της συνάρτησης. Ας μελετήσουμε την κλήση m(2).

Προκειμένου να υπολογιστεί η m(2) καλείται η f(m(1)). Κατά την κλήση της f(m(1)) πριν ο κώδικάς της αρχίσει να εκτελείται, θα πρέπει να υπολογιστεί η παράμετρός της, δηλαδή η m(1). Για αυτόν τον λόγο στον πίνακα 9.2 στη δεύτερη γραμμή βλέπουμε κλήση της m(1). Η m(1) υπολογίζεται ως f(m(0)). Κατά την κλήση της f(m(0)), πριν ο κώδικάς της αρχίσει να εκτελείται, θα πρέπει να υπολογιστεί η παράμετρός της, δηλαδή η m(0). Για αυτόν τον λόγο στον πίνακα 9.2 στην τρίτη γραμμή βλέπουμε κλήση της m(0). Η m(0) όμως είναι βασική περίπτωση και υπολογίζεται χωρίς περαιτέρω αναδρομική κλήση. Έτσι στη γραμμή 4 βλέπουμε πως η επιστροφή της m(0) είναι 0. Επομένως, η κλήση f(m(0)) έχει υπολογίσει την παράμετρό της και μπορεί να προχωρήσει στην εκτέλεση του σώματος της συνάρτησης. Έτσι στη γραμμή 5 βλέπουμε την f(0) η οποία επιστρέφει χωρίς περαιτέρω αναδρομική κλήση όπως φαίνεται στη γραμμή 6. Τώρα είναι εφικτός ο υπολογισμός της m(1) όπως φαίνεται στη γραμμή 7. Θυμηθείτε πως η m(2) που είναι το τελικό ζητούμενο είναι ίση με f(m(1)), εφόσον m(1)=0, καλείται η f(0) όπως φαίνεται στη γραμμή 8 που επιστρέφει 1 (γραμμή 9). Τέλος, η m(2) είναι ίση με 1. Παρόμοια, μπορείτε να αναλύσετε την κλήση f(2).

9.5 Πλεονεκτήματα και μειονεκτήματα

Πολλά προβλήματα μπορούν να λυθούν με αναδρομή ευκολότερα με πολύ μικρότερο και πιο κομψό κώδικα από ότι με μη αναδρομικές προσεγγίσεις. Για παράδειγμα, προσπαθήστε να κωδικοποιήσετε τους πύργους του Ανόι μη αναδρομικά και θα διαπιστώσετε πως πρόκειται για πρόβλημα σημαντικής δυσκολίας.

Επιπλέον, πολλά προβλήματα έχουν αναδρομικό ορισμό και επομένως η αναδρομική υλοποίησή τους είναι προφανής.

Από την άλλη πλευρά, η αναδρομή δεν είναι χωρίς μειονεκτήματα. Το σοβαρότερο από όλα είναι η πιθανή εξάντληση της στοίβας. Επομένως, θα πρέπει να χρησιμοποιείτε αναδρομή μόνο εκεί που γνωρίζετε πως τα μεγέθη των προβλημάτων είναι αδύνατο να εξαντλήσουν το μέγεθος της στοίβας. Οι προδιαγραφές της Java επιτρέπουν στοίβα σταθερού μεγέθους ή προσαρμοζόμενου. Εξαρτάται από την υλοποίηση της Java αν υποστηρίζει στοίβα σταθερού ή μεταβλητού μεγέθους. Στην πρώτη περίπτωση, η εξάντληση της στοίβας προκαλεί παραγωγή λάθους τύπου υπερχείλιση της στοίβας (stack overflow) και στη δεύτερη περίπτωση παραγωγή λάθους τύπου υπερχείλιση της μνήμης (memory overflow). Και στις δύο περιπτώσεις, το αποτέλεσμα για την εφαρμογή είναι το ίδιο, δηλαδή ο ανορθόδοξος τερματισμός της.

Επίσης, η αναδρομή σε αρκετές περιπτώσεις μπορεί να δίνει αποτελέσματα σε μεγαλύτερο χρονικό διάστημα από την μη αναδρομική λύση. Αν παρατηρήσετε το δέντρο των κλήσεων για τον υπολογισμό του Fibonacci(4) στο σχήμα 9.3 θα διαπιστώσετε ότι κάποιες κλήσεις επαναλαμβάνονται. Για παράδειγμα, η τιμή f(2) υπολογίζεται 2 φορές. Κατά τον υπολογισμό του εικοστού όρου της ακολουθίας, δηλαδή του f(20), η f(2) καλείται 4181 φορές. Επομένως σε αυτήν την περίπτωση, 4181 φορές επαναλαμβάνεται ο ίδιος υπολογισμός. Ωστόσο αυτό το πρόβλημα έρχεται να λύσει μια ειδική αναδρομική τεχνική γνωστή ως Απομνημόνευση που συζητάμε στην ενότητα 18.

Σε πολλές περιπτώσεις, οι εταιρείες παραγωγής λογισμικού υλοποιούν τη λύση ενός προβλήματος αρχικά αναδρομικά ώστε να βγουν εγκαίρως στην αγορά και στη συνέχεια αντικαθιστούν τη λύση με μη αναδρομική έκδοση. Σε κάθε περίπτωση, μια αναδρομική υλοποίηση μπορεί πάντα να μεταγραφεί με χρήση αποκλειστικά μη αναδρομικών τεχνικών.

Συμπερασματικά, χρησιμοποιήστε αναδρομή όπου σας διευκολύνει και εφόσον σας το επιτρέπει το μέγεθος του προβλήματος που έχετε να λύσετε.

9.6 Λυμένες Ασκήσεις

Σε αυτήν την ενότητα παρουσιάζουμε μια σειρά από παραδείγματα αναδρομικών συναρτήσεων ώστε να βοηθήσουμε τον αναγνώστη να εξοικειωθεί τόσο με την υλοποίηση όσο και με τη χρήση τους.

9.6.1 Παραγοντικό

Να αναπτυχθεί αναδρομική συνάρτηση που επιστρέφει το παραγοντικό της παραμέτρου της.

Λύση
Το παραγοντικό (factorial) ενός ακέραιου n είναι το γινόμενο όλων των θετικών ακέραιων που είναι μικρότεροι ή ίσοι του n. Το παραγοντικό του n συμβολίζεται σαν n!. Επομένως

$$n! = \begin{cases} 1 \times 2 \times 3 \ldots \times n, για\ n > 0 \\ 1, για\ n = 0 \end{cases}$$

Εκτός όμως από αυτόν τον ορισμό του παραγοντικού υπάρχει και ένας άλλος, αναδρομικός, ορισμός.

$$n! = \begin{cases} 1, για\ n = 0\ και\ n = 1 \\ n * (n-1)!, για\ n > 1 \end{cases}$$

Έχοντας τον αναδρομικό ορισμό είναι εύκολο να υλοποιήσουμε την αντίστοιχη συνάρτηση.

```
static int factorial(int n) { //Κώδικας 9.10
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return factorial(n - 1) * n;
    }
}
```

Κώδικας 9.10 *Υλοποίηση υπολογισμού του παραγοντικού με αναδρομή*

9.6.2 Ελάχιστο Κοινό Πολλαπλάσιο

Να αναπτυχθεί συνάρτηση αναδρομικού υπολογισμού του Ελάχιστου Κοινού Πολλαπλάσιου μιας σειράς ακέραιων. Στην main να αναπτυχθεί βασικός κώδικας ελέγχου.

Λύση
Το Ελάχιστο Κοινό Πολλαπλάσιο (ΕΚΠ) μιας σειράς ακέραιων, όπως φανερώνει το όνομά του, είναι το μικρότερο από τα κοινά πολλαπλάσια της σειράς. Υπάρχουν διάφοροι τρόποι υπολογισμού του. Ο τρόπος που χρησιμοποιούμε εδώ βασίζεται στο ακόλουθο σκεπτικό: Ο πρώτος υποψήφιος ακέραιος που μπορεί να είναι το ΕΚΠ της σειράς ακέραιων είναι ο μεγαλύτερος ακέραιος της σειράς. Αν ο ακέραιος αυτός δεν είναι το ΕΚΠ, τότε αναγκαστικά, ΕΚΠ θα είναι κάποιο πολλαπλάσιό του.

```
public class ElaxistoKoinoPollaplasio {

    private static int max(int[] nums) {
        int rVal = nums[0];
        for (int n : nums) {
            if (n > rVal) {
                rVal = n;
            }
        }
        return rVal;
    }

    private static boolean divideAll(int[] nums, int max) {
        for (int n : nums) {
            if (max % n != 0) {
                return false;
            }
        }
        return true;
    }
```

```
    public static int eKP(int... nums) { //variable length list of parameters
        return eKPR(1, nums);
    }

    private static int eKPR(int factor, int... nums) {
        int max = max(nums) * factor;
        if (divideAll(nums, max)) {
            return max;
        } else {
            return eKPR(++factor, nums);
        }

    }

    public static void main(String[] args) {
        System.out.println(eKP(24, 90, 60, 4, 89));
    }
}
```

Κώδικας 9.11 *Αναδρομικός υπολογισμός του ΕΚΠ σειράς ακέραιων*

Στον κώδικα 9.11 η αναδρομική συνάρτηση υπολογισμού του ΕΚΠ είναι η eKPR. Αυτή λαμβάνει ως παραμέτρους τον ακέραιο factor και μια σειρά ακέραιων. Επομένως, η δεύτερη παράμετρός της είναι μεταβλητού μήκους. Στην πρώτη γραμμή της συνάρτησης υπολογίζεται ο μεγαλύτερος ακέραιος της σειράς που είναι ο πρώτος υποψήφιος ακέραιος που πιθανώς είναι το ΕΚΠ. Ο μεγαλύτερος ακέραιος πολλαπλασιάζεται με την παράμετρο factor. Η παράμετρος factor διευκολύνει την αναδρομική κλήση. Κατά την αρχική κλήση της συνάρτησης, η factor πρέπει να έχει την τιμή 1, διαφορετικά ο υπολογισμός του ΕΚΠ πιθανότατα θα καταλήξει σε λάθος. Επομένως, κατά την πρώτη κλήση, η μεταβλητή max έχει την τιμή του μεγαλύτερου ακέραιου της σειράς. Προσέξτε πως η συνάρτηση max δεν επιστρέφει τη θέση του μεγαλύτερου ακεραίου αλλά την τιμή του. Στη συνέχεια ελέγχουμε αν ο max διαιρείται ακριβώς με όλους τους ακέραιους της σειράς. Αυτό επιτυγχάνεται με κλήση της divideAll. Αν η divideAll επιστρέψει true, το ΕΚΠ έχει βρεθεί και επιστρέφεται. Αν όμως επιστρέψει false, ο κώδικας προχωράει σε αναδρομική κλήση αυξάνοντας την factor κατά 1 ώστε να ελεγχθεί το πρώτο πολλαπλάσιο του μεγαλύτερου της σειράς. Στην επόμενη αναδρομική κλήση θα ελεγχθεί το επόμενο πολλαπλάσιο έως ότου βρεθεί ακέραιος που διαιρείται ακριβώς με όλους τους αριθμούς της σειράς εισόδου.

Ίσως να προσέξατε πως όλες οι συναρτήσεις στις οποίες αναφερθήκαμε είναι private, όπως και eKPR. Πράγματι, προκειμένου να κάνουμε την eKPR αναδρομική έχουμε προσθέσει την παράμετρο factor. Ο χρήστης της συνάρτησής μας, όταν θέλει να υπολογίσει το ΕΚΠ μιας σειράς ακέραιων, θα πρέπει να δίνει μόνο την σειρά ακέραιων. Δεν οφείλει να γνωρίζει πως η συνάρτηση είναι αναδρομική και πως κατά την κλήση της πρέπει να της περάσουμε και μια επιπλέον παράμετρο, ούτε τον ενδιαφέρει ο ρόλος αυτής της παραμέτρου. Σε αυτές τις περιπτώσεις κάνουμε την αναδρομική συνάρτηση ιδιωτική και παρέχουμε μια δημόσια συνάρτηση περικάλυμμα (wrapper). Στην περίπτωσή μας η συνάρτηση περιτύλιγμα είναι η eKP η οποία το μόνο που κάνει είναι να καλεί την eKPR με factor ίσο με 1.

9.6.3 Πρώτοι αριθμοί

Να αναπτυχθεί αναδρομική συνάρτηση που εξετάζει αν η παράμετρός της είναι πρώτος αριθμός. Στην main να αναπτυχθεί βασικός κώδικας ελέγχου.

Λύση

Πρώτος είναι ένας ακέραιος που διαιρείται ακριβώς μόνο με τον εαυτό του και το 1. Επιπλέον εξ ορισμού το 1 και το 2 δεν είναι πρώτοι. Επομένως, για να ελέγξουμε αν ένας ακέραιος, n, είναι πρώτος θα πρέπει να ελέγξουμε αν οποιοσδήποτε ακέραιος από το 2 έως n/2 διαιρεί ακριβώς τον n. Αν βρεθεί τέτοιος αριθμός, ο n δεν είναι πρώτος. Είναι προφανές πως κανένας ακέραιος μεγαλύτερος από n/2 δεν μπορεί να διαιρεί ακριβώς τον n.

```
public class PrimeNumber {
```

```
    private static boolean isPrime(int num, int divider) {
        if (num <= 2) {
            return false;
        }
        if (num % divider == 0) {
            return false;
        } else if (divider > num / 2) {
            return true;
        } else {
            return isPrime(num, divider + 1);
        }
    }

    public static boolean isPrime(int num) {
        return isPrime(num, 2);
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            System.out.println(i + " " + isPrime(i));
        }
    }
}
```

Κώδικας 9.12 *Αναδρομική αξιολόγηση πρώτου αριθμού*

Καταρχάς, στον κώδικα 9.12, στη συνάρτηση isPrime(int, int), ελέγχουμε αν ο υπό εξέταση αριθμός είναι μικρότερος ή ίσος από το 2, οπότε επιστρέφουμε false. Στη συνέχεια προκύπτουν δύο βασικές περιπτώσεις. Στην πρώτη περίπτωση, ο num διαιρείται ακριβώς με τον divider οπότε ο num δεν είναι πρώτος και επιστρέφουμε false. Στη δεύτερη βασική περίπτωση, ελέγχουμε αν ο divider υπερβαίνει το num/2. Σε αυτήν την περίπτωση έχουν ελεγχθεί όλοι οι ακέραιοι από 2 έως num/2 και κανένας δεν βρέθηκε να διαιρεί ακριβώς τον num, επομένως ο num είναι πρώτος και επιστρέφουμε true. Αν καμία από τις βασικές περιπτώσεις δεν προκαλέσει επιστροφή της isPrime, τότε προχωρούμε στην αναδρομική κλήση αυξάνοντας τον divider κατά 1.

Παρόμοια με την προηγούμενη άσκηση, ο divider ως παράμετρος εξυπηρετεί στην αναδρομική κλήση. Ο χρήστης της συνάρτησής μας δεν χρειάζεται να γνωρίζει καν την ύπαρξή του. Έτσι έχουμε κάνει την isPrime(int, int) private και παρέχουμε κατάλληλο περιτύλιγμα, την isPrime(int) που είναι public.

9.6.4 Selection Sort

Να αναπτυχθεί αναδρομική συνάρτηση ταξινόμησης μονοδιάστατου πίνακα ακέραιων με βάση τον αλγόριθμο selection sort (άσκηση 6.5.1). Να κατασκευαστεί κατάλληλος κώδικας ελέγχου με χρήση της Junit.

Λύση

Τη λογική του selection sort έχουμε ήδη παρουσιάσει στην άσκηση 6.5.1. Στον κώδικα 9.13 δίνουμε την αναδρομική και δομημένη υλοποίησή της.

```
import java.util.Arrays;

public class SelectionSort {//Κώδικας 9.13

    private static void swap(int[] array, int idx1, int idx2) {
        int i = array[idx1];
        array[idx1] = array[idx2];
        array[idx2] = i;
    }

    private static int idxOfMin(int[] array, int startIdx) {
        int min = array[startIdx];
        int rVal = startIdx;
```

```
        for (int i = startIdx + 1; i < array.length; i++) {
            if (array[i] < min) {
                min = array[i];
                rVal = i;
            }
        }
        return rVal;
    }

    private static void sort(int[] array, int startIdx) {
        int posOfMin = idxOfMin(array, startIdx);
        if (posOfMin != startIdx) {
            swap(array, posOfMin, startIdx);
        }
        startIdx++;
        if (startIdx < array.length - 1) {
            sort(array, startIdx);
        }
    }

    public static void sort(int[] array) {
        sort(array, 0);
    }

    public static void main(String[] args) {
        int[] tbl = {2, 1, 8, 6, 5, 4};
        sort(tbl);
        System.out.println(Arrays.toString(tbl));
    }
}
```

Κώδικας 9.13 *Αναδρομική υλοποίηση της selection sort*

Στον κώδικα 9.13 η αναδρομική υλοποίηση της selection sort δίνεται από τη συνάρτηση sort(int[], int). Η δεύτερη παράμετρος προκύπτει για διευκόλυνση της αναδρομικής κλήσης. Παρόμοια με τις δύο προηγούμενες ασκήσεις παρέχεται μια συνάρτηση-περιτύλιγμα, η sort(int[]). Οι υπόλοιπες συναρτήσεις είναι γνωστές καθώς έχουν ήδη συζητηθεί.

Ας δούμε λοιπόν την sort(int[], int). Η παράμετρος startIdx καθορίζει τη θέση του πίνακα από την οποία και μετά, ο πίνακας είναι αταξινόμητος. Αρχικά, η συνάρτηση εντοπίζει τη θέση του μικρότερου ακέραιου στο υπό ταξινόμηση τμήμα του πίνακα, δηλαδή από το startIdx και μετά. Αν το μικρότερο στοιχείο βρίσκεται μετά τη θέση startIdx, τότε ανταλλάσσει το μικρότερο στοιχείο με το στοιχείο στη θέση startIdx. Στη συνέχεια, αυξάνει το startIdx κατά 1 έτσι ώστε να συνεχίσει η ταξινόμηση από την επόμενη θέση στον πίνακα. Αν το startIdx δείχνει πριν την τελευταία θέση, σημαίνει ότι υπολείπεται τμήμα του πίνακα προς ταξινόμηση, οπότε γίνεται η αναδρομική κλήση, διαφορετικά τερματίζεται η αναδρομική κλήση και ο πίνακας είναι ταξινομημένος.

Στη συνέχεια παρουσιάζουμε τον κώδικα ελέγχου.

```
public class SelectionSortTest {

    public SelectionSortTest() {
    }

    /**
     * Test of sort method, of class SelectionSort.
     */
    @Test
    public void testSort() {
        System.out.println("sort");
        Random gen = new Random();
        for (int i = 0; i < 100; i++) {
```

```
            int[] array = new int[gen.nextInt(100) + 10];
            for (int j = 0; j < array.length; j++) {
                array[j] = gen.nextInt();
            }
            SelectionSort.sort(array);
            for (int j = 0; j < array.length - 1; j++) {
                if (array[j + 1] < array[j]) {
                    fail();
                }
            }
        }
    }

}
```

Κώδικας 9.14 *Κώδικας ελέγχου της SelectionSort.sort*

Στον κώδικα 9.14 κατασκευάζουμε 100 πίνακες με μέγεθος μεταξύ 10 και 110. Στη συνέχεια εκχωρούμε σε κάθε θέση ενός πίνακα έναν τυχαίο ακέραιο. Σημειώστε πως η nextInt χωρίς παραμέτρους μπορεί να επιστρέψει οποιαδήποτε ακέραιη τιμή. Αφού γεμίσουμε τον πίνακα με τυχαίες τιμές, προχωρούμε στην ταξινόμησή του. Τέλος, ελέγχουμε αν ο πίνακας είναι σωστά ταξινομημένος. Σε περίπτωση που εντοπίσουμε μικρότερο στοιχείο του πίνακα να έπεται μεγαλύτερου, καλούμε την fail και σημειώνουμε το λάθος.

9.6.5 Δυαδική αναζήτηση

Να αναπτυχθεί αναδρομική συνάρτηση δυαδικής αναζήτησης σε μονοδιάστατο πίνακα ακέραιων. Να κατασκευαστεί κατάλληλος κώδικας ελέγχου με χρήση της Junit.

Λύση
Τον αλγόριθμο δυαδικής αναζήτησης τον έχουμε ήδη συζητήσει και έχουμε δώσει και την μη αναδρομική υλοποίησή του (ενότητα 6.5.2). Έτσι, εδώ θα συζητήσουμε αποκλειστικά την αναδρομική υλοποίησή του.

```
public class BinarySearch {

    private static int binarySearch(int[] tbl, int schElement, int from, int to)
{
        if (from <= to) {
            int mid = from + (to - from) / 2;
            if (tbl[mid] == schElement) {
                return mid;
            }
            if (schElement < tbl[mid]) {
                return binarySearch(tbl, schElement, from, mid - 1);
            }
            return binarySearch(tbl, schElement, mid + 1, to);
        }
        return -1;
    }

    public static int binarySearch(int[] tbl, int schElement) {
        return binarySearch(tbl, schElement, 0, tbl.length - 1);
    }

}
```

Κώδικας 9.15 *Αναδρομική δυαδική αναζήτηση*

Στον κώδικα 9.15 η binarySearch(int[], int, int, int) είναι η αναδρομική συνάρτηση. Οι παράμετροι from και to έχουν προστεθεί για να διευκολύνουν την αναδρομή. Η συνάρτηση-περικάλυμμα καλεί την αναδρομική με παραμέτρους 0 και tbl.length-1 ώστε η αναζήτηση να αφορά καταρχάς όλον τον πίνακα. Μέσα στην

αναδρομική συνάρτηση γίνεται έλεγχος εάν το from είναι μικρότερο από το to, δηλαδή αν υπάρχει τμήμα του πίνακα που δεν έχει ελεγχθεί. Αν υπάρχει τέτοιο τμήμα, ενημερώνεται κατάλληλα η mid που δείχνει στο μέσο του υπό αναζήτηση τμήματος. Στη συνέχεια ελέγχεται αν το μέσον έχει την τιμή αναζήτησης. Αν ναι, η συνάρτηση επιστρέφει το μέσον ως τη θέση του στοιχείου που αναζητούμε. Αν όχι, η συνάρτηση προχωράει σε αναδρομική κλήση. Αν η τιμή του στοιχείου αναζήτησης είναι μικρότερη από την τιμή στο μέσο, η αναδρομική κλήση αφορά το πρώτο μισό του υπό εξέταση τμήματος, διαφορετικά αφορά το δεύτερο μισό.
Στη συνέχεια δίνουμε τον κώδικα ελέγχου.

```
import java.util.Random;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Lefteris Moussiades
 */
public class BinarySearchTest {

     /**
      * Test of binarySearch method, of class BinarySearch.
      */
     @Test
     public void testBinarySearch() {
         System.out.println("binarySearch");
         Random gen=new Random();
         for (int i=0; i<100; i++) {
             int[] array=new int[gen.nextInt(100)+10];
             for (int j=0; j<array.length; j++) {
                 array[j]=gen.nextInt();
             }
             SelectionSort.sort(array);
             int idx=gen.nextInt(array.length);
             idx=idx==array.length?array.length-1:idx;
             int schElement=array[idx];
             int result = BinarySearch.binarySearch(array, schElement);
             assertTrue(array[result]==schElement);
         }
     }
}
```

Κώδικας 9.16 *Έλεγχος της binarySearch*

Στον κώδικα 9.16 κατασκευάζουμε 100 πίνακες ακέραιων με τυχαίο μήκος από 10 έως 110 και τους γεμίζουμε με τυχαίες ακέραιες τιμές. Στη συνέχεια ταξινομούμε κάθε πίνακα και επιλέγουμε την τιμή ενός τυχαίου στοιχείου του. Τέλος καλούμε την binarySearch ώστε να αναζητήσουμε το στοιχείο που γνωρίζουμε ότι όντως υπάρχει στον πίνακα.

9.7 Ασκήσεις προς λύση

1. Να αναπτυχθεί συνάρτηση που υπολογίζει αναδρομικά τον Μέγιστο Κοινό Διαιρέτη σειράς ακέραιων. Να αναπτυχθεί κατάλληλος κώδικας ελέγχου.

2. Να αναπτυχθεί αναδρομική συνάρτηση static void prtSquareRoots(int i, int n) που τυπώνει τη σειρά των τετραγώνων των αριθμών από i έως n.

3. Να αναπτυχθεί αναδρομική συνάρτηση static void prtSquareRoots(int i, int n) που τυπώνει τη σειρά των τετραγώνων των αριθμών από i έως n, στην ίδια γραμμή, χωρισμένα με κόμμα και διάστημα. Μετά την εκτύπωση της σειράς, η συνάρτηση να προετοιμάζει ώστε η επόμενη έξοδος να βγαίνει στην επόμενη γραμμή. Για παράδειγμα, τα τετράγωνα των αριθμών από 0 έως 9 να τυπωθούν ως εξής:

```
0, 1, 4, 9, 16, 25, 36, 49, 64, 81
```

4. Να αναπτύξετε αναδρομική συνάρτηση υπολογισμού του αθροίσματος

$$\sum_{i=1}^{n} \frac{1}{i}$$

5. Πόσες αναδρομικές κλήσεις προκαλεί η κλήση fibonacci(9);

   Υπόδειξη: Τοποθετήστε την αναδρομική Fibonacci σε μία κλάση. Ορίστε στην κλάση μια ακέραια στατική μεταβλητή cnt. Αυξήστε την cnt κατάλληλα μέσα στην Fibonacci. Εμφανίστε το αποτέλεσμα.

6. Να αναπτυχθεί αναδρομική συνάρτηση που επιστρέφει τη μέγιστη τιμή ενός πίνακα ακέραιων.

7. Να αναπτυχθεί αναδρομική συνάρτηση που μετράει τους πεζούς χαρακτήρες σε έναν πίνακα χαρακτήρων.

8. Πόσες αναδρομικές κλήσεις απαιτεί η λύση των πύργων του Ανόι για 10 δίσκους;

9. Να αναπτυχθεί αναδρομική συνάρτηση που εμφανίζει όλες τις αναδιατάξεις (permutations) μιας συμβολοσειράς. Για παράδειγμα, έστω η συμβολοσειρά can, οι δυνατές αναδιατάξεις είναι can, cna, acn, anc, nca, nac.

10. Να αναπτυχθεί αναδρομική συνάρτηση που λαμβάνει ως παραμέτρους δύο συμβολοσειρές και εξετάζει αν η δεύτερη περιέχεται στην πρώτη.

11. Ένας τρόπος για τον υπολογισμό της τετραγωνικής ρίζας ενός πραγματικού αριθμού, x>0, έχει ως εξής: Έστω y μια τυχαία λύση. Αν το y υψωμένο στο τετράγωνο προσεγγίζει ικανοποιητικά το x, τότε η τετραγωνική ρίζα του x είναι το y, διαφορετικά αντικαθιστούμε στη θέση του y, το (y+x/y)/2. Υλοποιήστε αναδρομική συνάρτηση υπολογισμού της τετραγωνικής ρίζας πραγματικού αριθμού.

Βιβλιογραφία

[1] “Chapter 2. The Structure of the Java Virtual Machine.” https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5 (accessed Oct. 15, 2021).

[2] E. W. Weisstein, “Hofstadter Male-Female Sequences.” https://mathworld.wolfram.com/HofstadterMale-FemaleSequences.html (accessed Sep. 14, 2021).

Κριτήρια αξιολόγησης

Στατικές και τοπικές μεταβλητές

Έστω ο κώδικας

```
public class KA01 {

     static int i = 2;

     public static void main(String[] args) {
          for (int i = 5; i < 10; i++) {
               i++;
          }
          System.out.println(i--);
     }
}
```

Ποια είναι η έξοδός του;

```
A. 56789
B. 5678910
Γ. 2
Δ. 1
Ε. Λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Οι απαντήσεις Α και Β θα μπορούσαν να απασχολήσουν τον αναγνώστη, εφόσον μέσα στην επαναληπτική διαδικασία τυπώναμε το i. Εδώ το i απλώς αυξάνεται χωρίς να εμφανίζεται. Επιπλέον, η στατική μεταβλητή i είναι διαφορετική από την τοπική i της οποίας η εμβέλεια περιορίζεται στην for. Μεταβολές στην τοπική i δεν επηρεάζουν τη στατική i. Η απάντηση Δ είναι επίσης λανθασμένη. Παρότι η στατική i στέλνεται στην έξοδο συνοδευόμενη από τελεστή προσαύξησης, η προσαύξηση γίνεται αλλά το i-- αξιολογείται στην τιμή πριν την προσαύξηση καθώς χρησιμοποιείται ο μεταθεματικός τελεστής. Ο κώδικας δεν περιέχει λάθος μεταγλώττισης, επομένως και η απάντηση Ε είναι λανθασμένη. Τέλος, η σωστή απάντηση είναι η Γ.

Εμβέλεια τοπικών μεταβλητών

Έστω ο κώδικας

```
public class KA02 {

     public static void main(String[] args) {
          int i = 1;
          do {
               int j = i++;
               System.out.print(j);
          } while (j < 5);
     }
}
```

Ποια είναι η έξοδός του;

```
A. 1234
B. 2345
```

```
Γ. 12345
Δ. Λάθος χρόνου εκτέλεσης
Ε. Λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Το j είναι τοπική μεταβλητή στο μπλοκ της do. Επομένως η εμβέλειά του περιορίζεται σε αυτό το μπλοκ. Η συνθήκη ελέγχου στην while βρίσκεται εκτός του μπλοκ της do και αναφέρεται στο j. Το j όμως σε εκείνη την περιοχή του κώδικα είναι αθέατο στον μεταγλωττιστή ο οποίος παράγει κατάλληλο μήνυμα λάθους. Επομένως, η σωστή απάντηση είναι η E.

Μεταβλητή ελέγχου της for

Έστω ο κώδικας

```
public class KA03 {

     public static void main(String[] args) {
          int i = 5;
          for (i = 10; i > 0; i--) {
               System.out.print(--i);
          }
          System.out.println(i);
     }
}
```

Ποια είναι η έξοδός του;

```
Α. 975310
Β. 987654321
Γ. 97531
Δ. Λάθος χρόνου εκτέλεσης
Ε. Λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Η μεταβλητή ελέγχου της for ορίζεται μια φορά πριν τον βρόχο, οπότε μπορεί να χρησιμοποιηθεί χωρίς πρόβλημα. Έτσι, λάθος χρόνου εκτέλεσης δεν υπάρχει, άρα η απάντηση Δ δεν ισχύει. Επίσης, η απάντηση Ε δεν είναι σωστή, καθώς δεν υφίσταται κάποιο λάθος. Η απάντηση Β επίσης αποκλείεται, καθώς το i μειώνεται δύο φορές σε κάθε επαναληπτικό βήμα. Η απάντηση Γ είναι επίσης λανθασμένη. Η έξοδος 97531 πραγματοποιείται ως αποτέλεσμα του βρόχου. Ωστόσο, μετά το πέρας του βρόχου εκτελείται η System.out.println(i) η οποία τυπώνει το 0. Προσέξτε πως μετά την εκτύπωση του 1 από το σώμα της for, ο έλεγχος μεταβαίνει στην πρόταση της for όπου το i μειώνεται και γίνεται 0. Στη συνέχεια, η συνθήκη ελέγχου προκαλεί τερματισμό της for και ο κώδικας εκτελεί την τελική println με τιμή του i ίση με 0. Επομένως σωστή απάντηση είναι η Α.

Αρχικοποίηση τοπικών μεταβλητών

Έστω ο κώδικας

```
public class KA04 {
     public static void main(String[] args) {
          int x, y=5;
          if (y>5 || x>=Integer.MIN_VALUE) {
               x=y;
          }
          else {
```

```
                x=-y;
            }
            System.out.println(x*y);
        }
}
```

Ποια είναι η έξοδός του;

```
A. 25
B. -25
Γ. Λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Στη συνθήκη ελέγχου της if επιχειρείται προσπέλαση της x που είναι αναρχικοποίητη τοπική μεταβλητή. Αυτό έχει ως αποτέλεσμα την παραγωγή λάθους χρόνου μεταγλώττισης. Επομένως σωστή είναι η απάντηση Γ.

Πράξεις με ακέραιους

Έστω ο κώδικας

```
public class KA05 {
     public static void main(String[] args) {
            short k=1;
            k=k+1;
            int m=k++ + 1;
            System.out.println(m+k);
     }
}
```

Ποια είναι η έξοδός του;

```
A. 5
B. 6
Γ. Λάθος χρόνου μεταγλώττισης
Δ. 7
Ε. Λάθος χρόνου εκτέλεσης
```

Απάντηση/Λύση

Στη δεύτερη γραμμή της main επιχειρείται η εκχώρηση k=k+1. Παρότι το k είναι short, η έκφραση k+1 είναι τύπου int. Καθώς ο int είναι μεγαλύτερου μεγέθους από τον short, η εκχώρηση αυτή δεν είναι επιτρεπτή, οπότε η σωστή απάντηση είναι η Γ.

Switch 1

Έστω ο κώδικας

```
public class KA06 {
     public static void main(String[] args) {
            int x=1;
            int c=++x+1;
            switch (c) {
                  case 0 : System.out.println(0);
                  case 1 : System.out.println(1);
                  case x : System.out.println(x);
```

```
                case 3 : System.out.println(c);
            }
        }
}
```

Ποια είναι η έξοδός του;

```
Α. 0123
Β. 123
Γ. 23
Δ. 3
Ε. Λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Η τρίτη case της switch αναφέρεται στη μεταβλητή x. Στη θέση αυτή είναι αποδεκτές είτε κυριολεκτικές σταθερές είτε σταθερές μεταβλητές, δηλαδή μεταβλητές final. Η x δεν είναι ούτε το ένα, ούτε το άλλο. Επομένως, ο κώδικας παράγει λάθος χρόνου μεταγλώττισης και η σωστή απάντηση είναι η Ε.

Switch 2

Έστω ο κώδικας

```
public class KA07 {
     public static void main(String[] args) {
          int k=2;
          switch (k) {
               case 1: System.out.print(1);
               case 2: System.out.print(2);
               default: System.out.print(3);
          }
     }
}
```

Ποια είναι η έξοδός του;

```
Α. 1
Β. 2
Γ. 12
Δ. 3
Ε. 23
```

Απάντηση/Λύση

Η μεταβλητή k έχει την τιμή 2. Επομένως, η ροή του κώδικα κατευθύνεται στην case 2. Στη συνέχεια εκτελούνται οι προτάσεις μέχρι το τέλος της switch ή μέχρι το επόμενο break. Στο παράδειγμα αυτό δεν έχουμε break επομένως εμφανίζεται το 2 και στην συνέχεια το 3. Η σωστή απάντηση είναι η Ε.

Εκχώρηση και ισότητα

Έστω ο κώδικας

```
public class KA08 {
     public static void main(String[] args) {
          boolean b=7<=5;
          if (b=true) {
               System.out.println("7<=5");
```

```
            }
            else {
                  System.out.println("7>5");
            }
      }
}
```

Ποια είναι η έξοδός του;

```
A. 7<=5
B. 7>5
Γ. Λάθος χρόνου μεταγλώττισης
Δ. Λάθος χρόνου εκτέλεσης
```

Απάντηση/Λύση

Δεν υπάρχει λάθος χρόνου μεταγλώττισης, ούτε λάθος χρόνου εκτέλεσης. Προσέξτε πως στην if δεν χρησιμοποιείται ο τελεστής ισότητας αλλά ο τελεστής εκχώρησης. Έτσι το b δεν συγκρίνεται με το true αλλά εκχωρείται το true στο b. Η τιμή επιστροφής της εκχώρησης επομένως είναι true με αποτέλεσμα ο κώδικας να εισέρχεται στο μπλοκ της if και να εμφανίζει 7<=5. Η σωστή απάντηση επομένως είναι η Α.

Διαίρεση

Έστω ο κώδικας

```
public class KA09 {
      public static void main(String[] args) {
            System.out.println(3/2+""+3.0/2.0);
      }
}
```

Ποια είναι η έξοδός του;

```
A. 1.5 1.5
B. 1 1
Γ. 11.5
```

Απάντηση/Λύση

Το 2 και το 3 είναι κυριολεκτικές σταθερές ακέραιου τύπου. Επομένως και η διαίρεση μεταξύ τους είναι η ακέραια διαίρεση, δηλαδή 3/2=1. Αντίστοιχα, 3.0 και 2.0 είναι πραγματικές σταθερές και η μεταξύ τους διαίρεση είναι διαίρεση μεταξύ πραγματικών, δηλαδή 3.0/2.0=1.5. Δεδομένου ότι η προτεραιότητα της διαίρεσης είναι μεγαλύτερη από την προτεραιότητα της σύνδεσης-πρόσθεσης, η έξοδος του κώδικα είναι 11.5 και σωστή είναι η απάντηση Γ.

Διαίρεση και Σύνδεση

Έστω ο κώδικας

```
public class KA10 {
      public static void main(String[] args) {
            double i=1, j=0;
            System.out.print("sum equals to "+i+j);
            if (i/j> 1000) {
                  System.out.println("-Very Big");
            }
```

```
            else {
                  System.out.println("-Normal");
            }
      }
}
```

Ποια είναι η έξοδός του;

```
A. sum equals to 1-VeryBig
B. sum equals to 1-Normal
Γ. sum equals to 10-VeryBig
Δ. sum equals to 10-Normal
Ε. Λάθος χρόνου εκτέλεσης
ΣΤ. Τίποτε από τα παραπάνω
```

Απάντηση/Λύση

Η πρώτη print εμφανίζει sum equals to 10. Η έκφραση "sum equals to"+i+j αξιολογείται από αριστερά προς τα δεξιά. Η αλφαριθμητική σειρά "sum equals to " συνδέεται με το i το οποίο μετατρέπεται αυτόματα σε String. Το αποτέλεσμα "sum equals to 1" συνδέεται στη συνέχεια με το j. Στη συνέχεια, καθώς τα i και j είναι τύπου double, η διαίρεση i/j δεν προκαλεί λάθος χρόνου εκτέλεσης, αλλά επιστρέφει Infinite που θεωρείται μεγαλύτερο από το 1000, οπότε η συνθήκη της if αληθεύει και εμφανίζεται το -VeryBig. Επομένως, η σωστή απάντηση είναι η Γ.

Όρια Πίνακα 1

Έστω ο κώδικας

```
import java.util.Random;

public class KA11 {
      public static void main(String[] args) {
            Random g=new Random();
            int[] t={1,2,3,4,5,6};
            int idx=g.nextInt(6)+1;
            System.out.println(t[idx]);
      }
}
```

Ποια είναι η έξοδός του;

```
A. Τυπώνει έναν αριθμό από το 1 έως το 6
B. Τυπώνει έναν αριθμό από το 2 έως το 6
Γ. Λάθος χρόνου εκτέλεσης
Δ. Λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Η συνάρτηση nextInt(6) επιστρέφει έναν ακέραιο από το 0 έως το 5. Επομένως, οι πιθανές τιμές για τη μεταβλητή idx είναι από 1 έως 6. Παράλληλα το μήκος του πίνακα t είναι 6. Κατά συνέπεια, οι δείκτες του πίνακα λαμβάνουν τιμές στην περιοχή από 0 έως 5. Επομένως, αν η τιμή της idx είναι από 1 έως 5, η συνάρτηση θα τυπώσει έναν αριθμό από το 2 έως το 6. Ωστόσο, ο κώδικας δεν εγγυάται αυτές τις τιμές για το idx. Αντίθετα, όπως εξηγήσαμε παραπάνω, η τιμή 6 είναι πιθανή για την idx. Σε αυτήν την περίπτωση, στην τελευταία πρόταση επιχειρούμε να προσπελάσουμε έξω από τα όρια του πίνακα. Αυτό θα προκαλέσει λάθος χρόνου εκτέλεσης. Η σωστή απάντηση είναι η Γ.

Όρια Πίνακα 2

Έστω ο κώδικας

```
public class KA12 {

     public static void main(String[] args) {
          int[][] t2Dim = {
           {1, 2, 3, 4},
           {5, 6, 7, 8},
           {9, 10, 11, 12}
          };
          int[] t={1,0,2,3};
          for (int i=0; i<t.length; i+=2) {
               System.out.print(t2Dim[t[i]][t[i]]);
          }
     }
}
```

Ποια είναι η έξοδός του;

```
Α. 15
Β. 17
Γ. 19
Δ. 111
Ε. 611
ΣΤ. Λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Η for στην main ξεκινά από 0 και αυξάνεται με βήμα 2 έως ότου δεν υπερβεί το μήκος του πίνακα. Επομένως, η πρώτη τιμή της μεταβλητής i είναι το 0 και η δεύτερη το 2. Κατά συνέπεια, η t[i] για i=0 είναι ίση με 1 και η t[i] για i=2 είναι ίση με 2. Έτσι, ισχύει t2Dim[t[i]][t[i]]=t2Dim[1][1] για i=0 και t2Dim[t[i]][t[i]]=t2Dim[2][2]. Το μεν t2Dim[1][1] είναι ίσο με 6, το δε t2Dim[2][2] είναι ίσο με 11. Λάθος χρόνου μεταγλώττισης δεν υπάρχει, επομένως, η έξοδος είναι 611 και η σωστή απάντηση είναι η Ε.

Αναδρομή

Έστω ο κώδικας

```
public class KA13 {
     static void prt(int i) {
          System.out.print(i);
          if (i>=10)
               return;
          if (i%2==0) {
               prt(i+1);
          }
          else {
               prt(i+3);
          }
     }

     public static void main(String[] args) {
          prt(0);
     }
```

```
}
```

Ποια είναι η έξοδός του;

```
A. 01458912
B. 014589
Γ. 0347811
Δ. 03478
Ε. Καμία από τις παραπάνω
```

Απάντηση/Λύση

Κατά την κλήση prt(0) καταρχάς εμφανίζεται το 0. Στη συνέχεια, η συνθήκη ελέγχου i>=10 είναι ψευδής, ενώ η συνθήκη ελέγχου i%2==0 είναι αληθής, οπότε καλείται η prt(1). Η prt(1) αφού τυπώσει 1, καλεί την prt(4). Παρόμοια, η prt(4) εμφανίζει το 4 και καλεί prt(5). Εμφανίζεται το 5 και καλείται η prt(8). Εμφανίζεται το 8 και καλείται prt(9). Εμφανίζεται το 9 και καλείται η prt(12). Αυτή εμφανίζει πρώτα το 12 και στη συνέχεια βρίσκει false την έκφραση i>=10, οπότε τερματίζεται η αναδρομική κλήση. Καθώς από την επιστροφή από την αναδρομική κλήση, η prt δεν έχει άλλο κώδικα να εκτελέσει, η prt(0) τερματίζεται οριστικά. Επομένως, η έξοδος της prt(0) είναι 01458912 και σωστή απάντηση είναι η Α.

Αμοιβαία Αναδρομή

Έστω ο κώδικας

```
public class KA14 {

     static int f(int i) {
          if (i > 0) {
               return g(i);
          }
          return 0;
     }

     static int g(int i) {
          System.out.print(i);
          return f(i - 1);
     }

     public static void main(String[] args) {
          System.out.print(g(3));
     }
}
```

Ποιες από τις παρακάτω προτάσεις είναι σωστές;

```
A. Η έξοδος είναι 3210
B. Η g καλείται 3 φορές
Γ. Η f καλείται 3 φορές
Δ. Η έξοδος της κλήσης g(3) είναι 321
```

Απάντηση/Λύση

Η κλήση g(3) εμφανίζει 3 και καλεί την f(2). Η f(2) καλεί την g(2). Η g(2) εμφανίζει 2 και καλεί την f(1). Η f(1) καλεί την g(1) που εμφανίζει 1 και καλεί την f(0). Η f(0) επιστρέφει 0 και τερματίζει την αναδρομική κλήση. Επομένως, η g καλείται 3 φορές, το ίδιο και η f. Άρα οι προτάσεις Β και Γ είναι σωστές. Επιπλέον, η έξοδος της g(3) είναι 321, άρα και η πρόταση Δ είναι σωστή. Τέλος, η τιμή επιστροφής της g(3) είναι 0. Η

main εμφανίζει την τιμή αυτή, επομένως η έξοδος του κώδικα είναι 3210, δηλαδή και η πρόταση Α είναι σωστή.

Κλάση Math

Έστω ο κώδικας

```
public class KA15 {
     public static void main(String[] args) {
          float f;
          for (f=0; f<=9; f++); //γραμμή 4
          f=Math.sqrt(f) + 1;   //γραμμή 5
          f=f/1;
          System.out.println(f);
     }
}
```

Ποια από τις παρακάτω προτάσεις είναι σωστή;

```
Α. Η main έχει λάθος χρόνου μεταγλώττισης στη γραμμή 4
Β. Η main έχει λάθος χρόνου μεταγλώττισης στη γραμμή 5
Γ. Η main τυπώνει 4
Δ. Καμία από τις παραπάνω
```

Απάντηση/Λύση

Στη γραμμή 4 δεν υπάρχει λάθος. Όπως έχουμε εξηγήσει, κανένα από τα τμήματα της for δεν είναι υποχρεωτικό. Στη γραμμή 5, η sqrt επιστρέφει τιμή τύπου double, ενώ το f είναι float. Καθώς ο float είναι μικρότερης χωρητικότητας από τον double, η εκχώρηση αυτή δεν επιτρέπεται. Επομένως, η γραμμή 5 έχει λάθος χρόνου μεταγλώττισης. Κατά συνέπεια, ο κώδικας δεν μεταγλωττίζεται, άρα δεν τυπώνει 4. Η σωστή απάντηση είναι η Β.

Κλάση Boolean

Έστω ο κώδικας

```
public class KA16 {
     public static void main(String[] args) {
          int i=1, j=0;
          boolean b= i++>j && i>j+1; //2
          System.out.println(b                                    &&
          Boolean.parseBoolean("TRue"));
     }
}
```

Ποια είναι η έξοδός του;

```
Α. false
Β. true
Γ. Περιέχει λάθος χρόνου εκτέλεσης
```

Απάντηση/Λύση

Η έκφραση στη γραμμή 2 αξιολογείται ως εξής: Πρώτα αξιολογείται η έκφραση στα δεξιά της εκχώρησης, δηλαδή η έκφραση i++>j && i>j+1. Η αξιολόγηση εδώ γίνεται από αριστερά προς τα δεξιά. Η i++>j αυξάνει το i σε 2 και είναι αληθής, επομένως θα αξιολογηθεί και η i>j+1 η οποία επίσης είναι αληθής. Επομένως, η μεταβλητή b λαμβάνει την τιμή true. Η parseBoolean επιστρέφει επίσης true όταν το όρισμά της είναι η σειρά

true ανεξαρτήτως πεζών ή κεφαλαίων. Το λογικό and μεταξύ δύο εκφράσεων που είναι αληθείς επιστρέφει true. Λάθος χρόνου εκτέλεσης δεν υπάρχει. Συνεπώς, η main εμφανίζει true και η σωστή απάντηση είναι η Β.

Συστήματα Αρίθμησης

Έστω ο κώδικας

```
public class KA17 {
     public static void main(String[] args) {
          int i=010, j=011;
          System.out.println(i+j);
     }
}
```

Ποια είναι η έξοδός του;

```
Α. 21
Β. 17
Γ. Περιέχει λάθος χρόνου εκτέλεσης
Δ. Περιέχει λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Όπως έχουμε αναφέρει στην ενότητα 3.6, οι κυριολεκτικές ακέραιες σταθερές που αρχίζουν από μηδέν εκφράζονται στο οκταδικό σύστημα. Ο 010 είναι η βάση του οκταδικού συστήματος, δηλαδή το 8. Εύκολα, μπορεί κανείς να καταλάβει πως ο 011 είναι ο επόμενος ακέραιος από τον 010, δηλαδή το 9. Επομένως, η έξοδος του κώδικα είναι 17 και η σωστή απάντηση η Β.

Κεφάλαιο 10

Σύνοψη
Καταρχάς γίνεται μια σύντομη ιστορική αναδρομή της ανάπτυξης του αντικειμενοστρεφούς μοντέλου. Στη συνέχεια παρουσιάζονται τα θεμελιώδη χαρακτηριστικά του αντικειμενοστρεφούς μοντέλου, η Ενσωμάτωση (Encapsulation), η Κληρονομικότητα (Inheritance), ο Πολυμορφισμός (Polymorphism) και η Αφαιρετικότητα (Abstraction). Συζητείται η αναλογία μεταξύ των θεμελιωδών χαρακτηριστικών και των προβλημάτων του φυσικού κόσμου και εξηγούνται οι τρόποι με τους οποίους το αντικειμενοστρεφές μοντέλο διευκολύνει στη μοντελοποίηση των προβλημάτων του φυσικού κόσμου.

Προαπαιτούμενη γνώση
Η εισαγωγική αυτή ενότητα προϋποθέτει γνώση του διαδικαστικού προγραμματισμού με Java.

Λέξεις κλειδιά
Αντικειμενοστρεφής προγραμματισμός (Object Oriented Programming), Διαδικαστικός Προγραμματισμός (Procedural Programming), Ενσωμάτωση (Encapsulation), Κληρονομικότητα (Inheritance), Πολυμορφισμός (Polymorphism), Αφαιρετικότητα (Abstraction).

10 Εισαγωγή στο Αντικειμενοστρεφές μοντέλο

Το αντικειμενοστρεφές μοντέλο αντλεί την καταγωγή του από την εφαρμογή Sketchpad [1] του Ivan Sutherland που δημιουργήθηκε στις αρχές της δεκαετίας του 1960. Η Sketchpad υποστήριζε κληρονομικότητα βασιζόμενη σε πρωτότυπα με πολλά κοινά στοιχεία με τη σημερινή Javascript και δυναμική διασύνδεση, βασικό χαρακτηριστικό των περισσοτέρων γλωσσών αντικειμενοστρεφούς προγραμματισμού, σήμερα. Την ίδια περίοδο έκανε την εμφάνισή της η AED-0, μια έκδοση της Algol που υποστήριζε την άμεση διασύνδεση δεδομένων και συναρτήσεων που αποτελεί θεμελιώδες χαρακτηριστικό του αντικειμενοστρεφούς μοντέλου. Το 1965 εμφανίστηκε η Simula [2], η πρώτη γλώσσα προγραμματισμού που στη συνέχεια αναγνωρίστηκε ευρέως ως αντικειμενοστρεφής. Ωστόσο, τον όρο «Αντικειμενοστρεφής Προγραμματισμός» επινόησε ο Alan Kay το 1967.

Τη δεκαετία του 1970 αναπτύχθηκε η Smalltalk [3] που εισήγαγε την έννοια των κλάσεων από τους Alan Kay, Dan Ingalls, Adele Goldberg και άλλους στην Xerox PARC.

Στα μέσα του 1980 αναπτύχθηκε η Objective-C που προσέθεσε στην C στοιχεία από την SmallTalk και αποτέλεσε μέχρι το 2014, οπότε αντικαταστάθηκε από την Swift, την κύρια γλώσσα προγραμματισμού για τα λειτουργικά macOS και iOS της Apple.

Ήδη από το 1979, ο Bjarne Stroustrup, άρχισε την ανάπτυξη μιας προέκτασης της C που ονόμασε C with classes, ενώ το 1982 άρχισε την ανάπτυξη της C++ [4], η αρχική έκδοση της οποίας βγήκε το 1985.

Τη δεκαετία του 1990, το αντικειμενοστρεφές μοντέλο έγινε το κυρίαρχο μοντέλο προγραμματισμού, καθώς οι αντικειμενοστρεφείς γλώσσες διαδόθηκαν ευρέως.

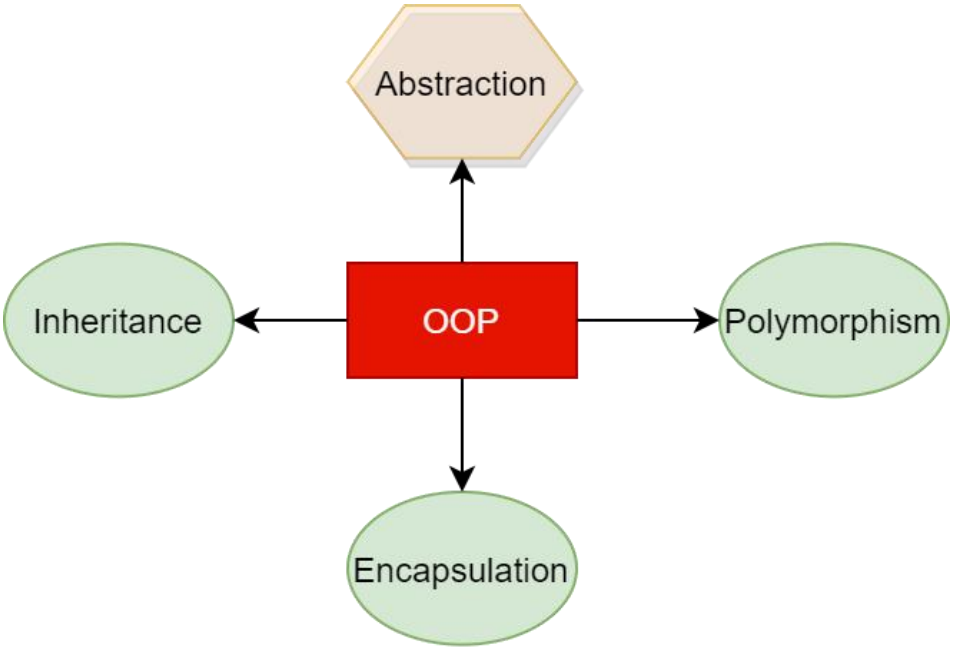
Η πρώτη έκδοση της Python κυκλοφόρησε το 1994 με δημιουργό τον Van Rossum. Με την εμφάνισή της, η Python ενσωματώνει τα βασικά χαρακτηριστικά του αντικειμενοστρεφούς μοντέλου τα οποία συνδυάζει με το λειτουργικό (functional) μοντέλο. Λίγο αργότερα, το 1996, κυκλοφόρησε και η πρώτη έκδοση της Java το 1996 από την Sun Microsystems.

Πέραν των προαναφερόμενων γλωσσών, εμφανίστηκαν και αρκετές άλλες ενώ χαρακτηριστικά του αντικειμενοστρεφούς μοντέλου, προστέθηκαν σε αρκετές προϋπάρχουσες.

Πρόσφατα, πολλές αντικειμενοστρεφείς γλώσσες, συμπεριλαμβανομένης της Java, ενσωματώνουν χαρακτηριστικά του λειτουργικού (functional) προγραμματισμού, ενισχύοντας έτσι την αποτελεσματικότητά τους.

10.1 Τα θεμελιώδη χαρακτηριστικά του Αντικειμενοστρεφούς Προγραμματισμού

Το αντικειμενοστρεφές μοντέλο σχεδιάστηκε ώστε να εξυπηρετεί δύο βασικούς στόχους. Ο πρώτος αφορά στη διευκόλυνση της μοντελοποίησης των προβλημάτων του πραγματικού κόσμου και ο δεύτερος στον έλεγχο των λαθών που όχι σπάνια υπεισέρχονται στους κώδικες εφαρμογών.
Τους στόχους του, το αντικειμενοστρεφές μοντέλο επιτυγχάνει βασιζόμενο στα θεμελιώδη χαρακτηριστικά του: την Ενσωμάτωση (Encapsulation), την Κληρονομικότητα (Inheritance) και τον Πολυμορφισμό (Polymorphism). Στο σχήμα 10.1, παρουσιάζονται τα θεμελιώδη χαρακτηριστικά του αντικειμενοστρεφούς μοντέλου. Στο σχήμα συμπεριλαμβάνεται και η Αφαιρετικότητα (Abstraction) που παρουσιάζεται παρακάτω σε αυτήν την ενότητα.

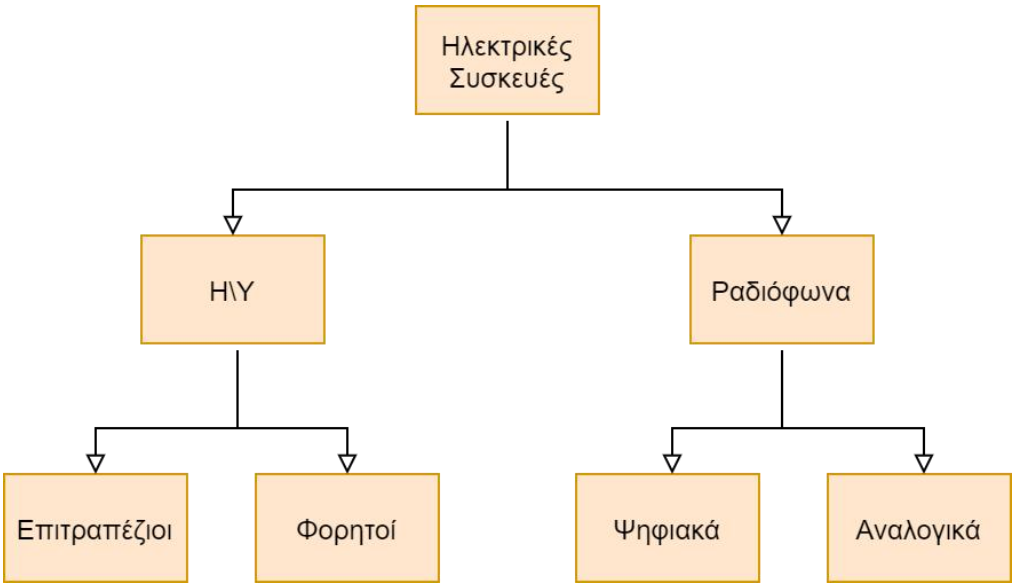


Σχήμα 10.1 *Τα θεμελιώδη χαρακτηριστικά του αντικειμενοστρεφούς μοντέλου*

Ο πραγματικός κόσμος αποτελείται από αντικείμενα, π.χ. ένας Ηλεκτρονικός Υπολογιστής, ένα βιβλίο, μια καρέκλα. Τα αντικείμενα συχνά διαθέτουν ιδιότητες και επιτελούν λειτουργίες. Για παράδειγμα, ένας Ηλεκτρονικός Υπολογιστής διαθέτει ιδιότητες όπως η χωρητικότητα της μνήμης RAM, ο τύπος του επεξεργαστή του κ.ά. Επίσης, επιτελεί λειτουργίες, όπως εκκίνηση λειτουργίας, τερματισμός, κ.ά. Το αντικειμενοστρεφές μοντέλο μας δίνει τη δυνατότητα να ορίσουμε οντότητες μέσα στον κώδικά μας που ενσωματώνουν ιδιότητες και λειτουργίες και να αναπαραστήσουμε έτσι τα αντικείμενα του πραγματικού κόσμου με ευκολία. Πρόκειται για την πιο βασική αρχή του αντικειμενοστρεφούς προγραμματισμού, την Ενσωμάτωση. Λεπτομέρειες για την Ενσωμάτωση θα βρείτε στην ενότητα 11.

Τα αντικείμενα του πραγματικού κόσμου είναι οργανωμένα σε κατηγορίες. Για παράδειγμα, ο δικός μου υπολογιστής ανήκει στην κατηγορία των επιτραπέζιων υπολογιστών, ο φορητός υπολογιστής της συζύγου μου ανήκει στην κατηγορία των φορητών υπολογιστών, το ραδιόφωνό που έχουμε στο σαλόνι ανήκει στην κατηγορία των αναλογικών ραδιοφώνων και αυτό που έχουμε στην κρεβατοκάμαρα στην κατηγορία των ψηφιακών ραδιοφώνων. Στα πλαίσια του αντικειμενοστρεφούς μοντέλου, οι κατηγορίες αντικειμένων αναφέρονται συνήθως με τον όρο κλάσεις (classes). Έτσι, έχουμε την κλάση των επιτραπέζιων υπολογιστών και την κλάση των φορητών υπολογιστών, την κλάση των αναλογικών ραδιοφώνων και την κλάση των ψηφιακών ραδιοφώνων. Προσέξτε όμως πως οι επιτραπέζιοι και οι φορητοί υπολογιστές συνιστούν υποκατηγορίες ή υποκλάσεις (subclasses) της κλάσης των ηλεκτρονικών υπολογιστών. Αντίστοιχα, η κλάση των ηλεκτρονικών υπολογιστών είναι υπερκλάση (superclass) της κλάσης των επιτραπέζιων και της κλάσης των φορητών υπολογιστών. Παρόμοια, τα αναλογικά και τα ψηφιακά ραδιόφωνα αποτελούν υποκλάσεις της γενικότερης κλάσης των ραδιοφώνων. Επιπλέον, τόσο τα ραδιόφωνα όσο και οι ηλεκτρονικοί υπολογιστές είναι υποκλάσεις της κλάσης των ηλεκτρικών συσκευών.

Γενικότερα, τα αντικείμενα του πραγματικού κόσμου οργανώνονται σε κλάσεις οι οποίες σχετίζονται μεταξύ τους με ιεραρχικές σχέσεις. Στο σχήμα 10.2 απεικονίζονται οι ιεραρχικές σχέσεις των κλάσεων που αναφέραμε ως παραδείγματα.



Σχήμα 10.2 *Ιεραρχικές σχέσεις μεταξύ των κλάσεων*

Οι ηλεκτρικές συσκευές διαθέτουν μια λειτουργία που τους επιτρέπει να συνδέονται ή να αποσυνδέονται στο ηλεκτρικό ρεύμα. Τη λειτουργία αυτή όμως διαθέτουν και οι Η\Υ και τα ραδιόφωνα καθώς τόσο οι Η\Υ όσο και τα ραδιόφωνα είναι ηλεκτρικές συσκευές. Το ίδιο ισχύει για τις κλάσεις των επιτραπέζιων και φορητών υπολογιστών, των ψηφιακών και των αναλογικών ραδιοφώνων. Γενικότερα, μια λειτουργία που ορίζεται σε μια υπερκλάση αποτελεί επίσης λειτουργία κάθε υποκλάσης της. Το αντικειμενοστρεφές μοντέλο μέσα από τη δεύτερη θεμελιώδη αρχή του, την Κληρονομικότητα, μας δίνει τη δυνατότητα να ορίζουμε μια ιδιότητα ή λειτουργία σε μια κλάση και στη συνέχεια η λειτουργία/ιδιότητα κληρονομείται αυτόματα σε όλες τις υποκλάσεις της. Το χαρακτηριστικό αυτό διευκολύνει την εφαρμογή της επαναχρησιμοποίησης κώδικα (code reusability) και διαμορφώνει το μοντέλο για την εφαρμογή του τρίτου θεμελιώδους χαρακτηριστικού του, του Πολυμορφισμού. Λεπτομέρειες για την Κληρονομικότητα θα βρείτε στην ενότητα 13.

Όπως αναφέρθηκε ήδη, ο υπολογιστής μου είναι επιτραπέζιος, επομένως ανήκει στην κλάση των επιτραπέζιων υπολογιστών. Ταυτόχρονα όμως είναι ένας (is a) Η\Υ αλλά και μια ηλεκτρική συσκευή. Σε αντιστοιχία με τον πραγματικό κόσμο, το αντικειμενοστρεφές μοντέλο μας δίνει τη δυνατότητα να δημιουργήσουμε ένα αντικείμενο τύπου επιτραπέζιου υπολογιστή και ανάλογα με τις ανάγκες του κώδικά μας να το διαχειριζόμαστε σαν Η\Υ ή σαν ηλεκτρική συσκευή, δηλαδή σαν να είναι τύπου μιας υπερκλάσης. Αυτή η δυνατότητα συνιστά τη βάση του Πολυμορφισμού και είναι πολύ σημαντική όπως θα δούμε στην ενότητα 13 που εξετάζουμε αναλυτικά τον πολυμορφισμό.

Πολλοί συγγραφείς θεωρούν ένα τέταρτο θεμελιώδες χαρακτηριστικό του αντικειμενοστρεφούς προγραμματισμού, την Αφαιρετικότητα (Abstraction) [1,4]. Αφαιρετικότητα σημαίνει απόκρυψη των μη αναγκαίων πληροφοριών από τον χρήστη του κώδικα. Ωστόσο, άλλοι συγγραφείς δεν θεωρούν την αφαιρετικότητα ως θεμελιώδες χαρακτηριστικό του αντικειμενοστρεφούς μοντέλου [2,3]. Παρότι υπάρχει καθολική συμφωνία πως η αφαιρετικότητα είναι ένα χαρακτηριστικό του αντικειμενοστρεφούς προγραμματισμού, προκύπτει διχογνωμία για το κατά πόσο αποτελεί θεμελιώδες γνώρισμά του. Οι συγγραφείς που δεν θεωρούν την αφαιρετικότητα ως θεμελιώδες χαρακτηριστικό, επιχειρηματολογούν υποστηρίζοντας πως αυτή συναντάται και στο διαδικαστικό μοντέλο επάνω στο οποίο στηρίζεται το αντικειμενοστρεφές. Επίσης, κάθε πληροφοριακό σύστημα, σωστά οργανωμένο, χαρακτηρίζεται από κάποιο βαθμό αφαιρετικότητας. Επιπλέον, παρότι η αφαιρετικότητα διαθέτει και τους δικούς της ιδιαίτερους μηχανισμούς σε αρκετές γλώσσες και στην Java, είναι ταυτόχρονα αναγκαίο αποτέλεσμα της ενσωμάτωσης, της κληρονομικότητας και του πολυμορφισμού. Με αυτά τα δεδομένα, τείνουν να θεωρούν την αφαιρετικότητα ως παράγωγο παρά ως θεμελιώδες γνώρισμα.

Βιβλιογραφία

[1] “Sketchpad,” Wikipedia. Dec. 28, 2020. Accessed: Feb. 10, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Sketchpad&oldid=996705015

[2] J. R. Holmevik, “Compiling SIMULA: a historical study of technological genesis,” IEEE Annals of the History of Computing, vol. 16, no. 4, pp. 25–37, Winter 1994, doi: 10.1109/85.329756.

[3] “The Early History of Smalltalk,” Jul. 10, 2008. https://web.archive.org/web/20080710144930/http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html (accessed Feb. 10, 2021).

[4] “Stroustrup: The C++ Programming Language.” https://www.stroustrup.com/1st.html (accessed Feb. 10, 2021).

Κεφάλαιο 11

Σύνοψη
Σε αυτήν την ενότητα παρουσιάζεται η Ενσωμάτωση, το θεμελιώδες χαρακτηριστικό του αντικειμενοστρεφούς μοντέλου με βάση το οποίο οι συναρτήσεις συνδέονται άμεσα με τα δεδομένα στα οποία επενεργούν. Εξηγούνται αναλυτικά τα πλεονεκτήματα της Ενσωμάτωσης σχετικά με τη μοντελοποίηση του φυσικού κόσμου και τον περιορισμό των λαθών στις εφαρμογές. Παρουσιάζονται και εξηγούνται οι αναγκαίες συναρτήσεις μια κλάσης σύμφωνα με το μοντέλο της Java.

Προαπαιτούμενη γνώση
Το διαδικαστικό μοντέλο προγραμματισμού με Java

Λέξεις κλειδιά
Ενσωμάτωση (Encapsulation), αντικείμενο (object), κλάση (class), στιγμιότυπο (instance), μεταβλητή στιγμιότυπου (instance variable), συνάρτηση στιγμιότυπου (instance function), μέθοδος (method), δημιουργός (constructor), αναγνώστης (getter), ρυθμιστής (setter), προκαθορισμένη προσπέλαση ή προσπέλαση πακέτου (default access ή package access)

11 Ενσωμάτωση

Ας υποθέσουμε ότι θέλουμε να αναπτύξουμε μια εφαρμογή στην οποία απαιτούνται υπολογισμοί με διάφορα γεωμετρικά σχήματα, π.χ. χρειάζεται να υπολογίσουμε την περίμετρο ενός ορθογώνιου παραλληλόγραμμου, το εμβαδόν ενός κύκλου, κ.ά. Στο πλαίσιο του διαδικαστικού μοντέλου θα αναπτύξουμε κατάλληλη βιβλιοθήκη συναρτήσεων και στην εφαρμογή μας θα καλούμε τις συναρτήσεις της βιβλιοθήκης. Ο τρόπος για να υλοποιήσουμε μια τέτοια βιβλιοθήκη στην Java είναι η ανάπτυξη μιας ή περισσοτέρων κλάσεων που θα περιλαμβάνουν στατικές συναρτήσεις.

Στον κώδικα 11.1, παρουσιάζουμε μια τέτοια κλάση. Για απλοποίηση θεωρούμε ότι μας ενδιαφέρει μόνο το εμβαδόν, η περίμετρος και η διαγώνιος του ορθογώνιου παραλληλόγραμμου.

```
public class Lib {

    public static double rectArea(double length, double width) {
        return length * width;
    }

    public static double rectPerimeter(double length, double width) {
        return 2 * (length + width);
    }

    public static double rectDiagonal(double length, double width) {
        return Math.sqrt(Math.pow(length, 2) + Math.pow(width, 2));
    }
}
```

Κώδικας 11.1 *Βιβλιοθήκη στατικών συναρτήσεων γεωμετρικών υπολογισμών*

Στη συνέχεια στον κώδικα-πελάτη (client code), δηλαδή στην εφαρμογή μας θα χρησιμοποιήσουμε τις συναρτήσεις της Lib για να κάνουμε τους απαιτούμενους υπολογισμούς, όπως φαίνεται στον κώδικα 11.2. Σημειώστε πως για να χρησιμοποιήσετε την Lib θα πρέπει να την κάνετε import.

```
public class App {

    public static void main(String[] args) {
        double rectA = 6, rectB = 8;
        System.out.println(rectArea(rectA, rectB));
        System.out.println(rectPerimeter(rectA, rectB));
        System.out.println(rectDiagonal(rectA, rectB));
```

```
        }
    }
```

Κώδικας 11.2 *Εφαρμογή που χρησιμοποιεί τις συναρτήσεις της Lib*

Όπως δείχνει ο κώδικας 11.2, η εφαρμογή καλεί τις συναρτήσεις βιβλιοθήκης, περνώντας ως πραγματικές παραμέτρους τις μεταβλητές rectA και rectB που αναπαριστούν το μήκος και το πλάτος ενός ορθογώνιου. Ο μοναδικός έλεγχος από τον μεταγλωττιστή σε αυτήν τη φάση αφορά στη συμβατότητα τύπου μεταξύ πραγματικών και τυπικών παραμέτρων. Τι θα συμβεί όμως αν ο προγραμματιστής της εφαρμογής περάσει ως παράμετρο μια μεταβλητή τύπου double που δεν αναπαριστά ούτε μήκος, ούτε πλάτος ορθογώνιου; Τι θα συμβεί αν περάσει ως παραμέτρους μεταβλητές που αναπαριστούν το μήκος μιας πλευράς ενός ορθογώνιου και το μήκος πλευράς άλλου ορθογώνιου; Τι θα συμβεί αν περάσει μια αναρχικοποίητη μεταβλητή; Σε όλες αυτές τις περιπτώσεις θα έχει εισαχθεί ένα λάθος χρόνου εκτέλεσης στην εφαρμογή. Μπορεί να σκεφτείτε πως το πρόβλημα δεν είναι πολύ σοβαρό, καθώς έχουμε όλες κι όλες 3 μεταβλητές, οι δύο εκφράζουν το μήκος και το πλάτος ενός ορθογώνιου και η τρίτη, η radius, ενδεχομένως την ακτίνα ενός κύκλου. Επομένως είναι κάπως δύσκολο ο προγραμματιστής εφαρμογών να κάνει λάθος. Κάτι τέτοιο φαίνεται να ισχύει για το παράδειγμά μας. Σκεφτείτε όμως έναν ρεαλιστικό κώδικα, με χιλιάδες γραμμές, με χιλιάδες μεταβλητές στον οποίο κάποιες μεταβλητές υπολογίζονται κατά τον χρόνο εκτέλεσης. Ένα τέτοιο περιβάλλον αυξάνει σημαντικά την πιθανότητα λαθών τα οποία όμως είναι πολύ σοβαρά και συχνά οδηγούν σε απρόβλεπτες συνέπειες με μεγάλο κόστος. Επομένως, δεν είναι καλή ιδέα, η αποφυγή τους να επαφίεται αποκλειστικά στην ευθύνη του προγραμματιστή εφαρμογών. Ωστόσο, το διαδικαστικό μοντέλο δεν μπορεί να δώσει ικανοποιητικές απαντήσεις στο πρόβλημά μας. Αυτό οφείλεται στο ότι το μοντέλο διατηρεί χωριστά τις λειτουργίες, δηλαδή τις συναρτήσεις από τα δεδομένα στα οποία αυτές επενεργούν. Ικανοποιητική απάντηση στα ερωτήματα μας έρχεται να δώσει το αντικειμενοστρεφές μοντέλο με την Ενσωμάτωση (Encapsulation).

Με την Ενσωμάτωση, συναρτήσεις και δεδομένα συνδυάζονται σε μια οντότητα, το λεγόμενο αντικείμενο (object). Σε σχέση με το παράδειγμά μας, μπορούμε να δημιουργούμε αντικείμενα που αναπαριστούν ορθογώνια στα οποία τα δεδομένα, δηλαδή το μήκος των πλευρών συνδέεται με τις συναρτήσεις υπολογισμού της περιμέτρου, του εμβαδού και της διαγωνίου. Σε αυτήν την περίπτωση δεν απαιτείται να περάσουμε ως παραμέτρους στις εν λόγω συναρτήσεις μεταβλητές που αναπαριστούν τα μήκη των πλευρών, οπότε αυτομάτως περιορίζεται ο κίνδυνος λαθών. Προκειμένου όμως να είμαστε σε θέση να δημιουργήσουμε τέτοια αντικείμενα θα πρέπει πρώτα να ορίσουμε την κλάση τους. Στον κώδικα 11.3 δίνεται ένας αρχικός ορισμός της κλάσης των ορθογώνιων παραλληλόγραμμων.

```
public class Rectangle {

    public double length, width;

    public double area() {
        return length * width;
    }

    public double perimeter() {
        return 2 * (length + width);
    }

    public double diagonal() {
        return Math.sqrt(Math.pow(length, 2) + Math.pow(width, 2));
    }
}
```

Κώδικας 11.3 *Πρώτη έκδοση της κλάσης Rectangle*

Η κλάση ορίζεται με τη χρήση της λέξης-κλειδί class ακολουθούμενης από το όνομα που δίνουμε στην κλάση. Στη δική μας περίπτωση διαλέξαμε το όνομα Rectangle. Προσέξτε πως οι μεταβλητές length και width είναι μη στατικές (non-static). Οι μεταβλητές αυτές ονομάζονται και μεταβλητές στιγμιότυπου (instance variables). Επίσης οι μη στατικές συναρτήσεις area, perimeter και diagonal ονομάζονται συναρτήσεις στιγμιότυπου (instance functions). Επιπλέον, οι συναρτήσεις μιας κλάσης ονομάζονται και μέθοδοι (methods). Ας δούμε όμως πώς δημιουργούμε και αξιοποιούμε τα αντικείμενα της κλάσης Rectangle.

```
public class App2 {

    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.length = 2d;
        r.width = 2d;
        System.out.println(r);
        System.out.println(r.area());
        System.out.println(r.perimeter());
        System.out.println(r.diagonal());
    }
}
```

Κώδικας 11.4 *Εφαρμογή που αξιοποιεί την κλάση Rectangle*

Στην πρώτη γραμμή της main δηλώνουμε μια μεταβλητή r τύπου Rectangle. Παράλληλα δημιουργούμε ένα Rectangle με χρήση της λέξης-κλειδί new. Επομένως, η μεταβλητή r είναι μεταβλητή αναφοράς. Το new συνοδεύεται από το Rectangle() που αποτελεί τον προκαθορισμένο δημιουργό (default constructor) της κλάσης. Για τους δημιουργούς (constructors) συζητάμε αναλυτικά παρακάτω σε αυτήν την ενότητα. Για την ώρα αρκεί να γνωρίζουμε πως η κλήση στην πρώτη γραμμή της main δημιουργεί ένα αντικείμενο Rectangle και εκχωρεί τη διεύθυνσή του στη μεταβλητή r. Το αντικείμενο (object) που δημιουργείται ονομάζεται και στιγμιότυπο (instance). Κατά τη δημιουργία του στιγμιότυπου, κάθε bit μη στατικής μεταβλητής αρχικοποιείται στο 0. Επομένως, τα πεδία length και width αρχικοποιούνται αυτόματα στην τιμή 0d. Στη συνέχεια, η εφαρμογή προσπελαύνει τα length και width και τους δίνει τιμές. Όπως φαίνεται, η προσπέλαση γίνεται με χρήση του τελεστή τελεία (dot operator) και της μεταβλητής r που αναφέρεται στο συγκεκριμένο στιγμιότυπο. Ας τονιστεί εδώ πως κάθε στιγμιότυπο διαθέτει δικές του μεταβλητές στιγμιότυπου. Αν για παράδειγμα δημιουργήσουμε ένα ακόμη Rectangle, ας πούμε r1, τότε για τις μεταβλητές length και width του r1, θα δεσμευτεί επιπλέον χώρος στη μνήμη για δύο double. Επομένως κάθε στιγμιότυπο Rectangle "γνωρίζει" τις διαστάσεις του και επάνω σε αυτές επενεργούν οι συναρτήσεις του. Είναι φανερό πως αυτή η οργάνωση συνδέει άμεσα τις συναρτήσεις με τις μεταβλητές ενός στιγμιότυπου, απαλλάσσοντας έτσι τον προγραμματιστή της εφαρμογής από αυτήν την ευθύνη. Αυτός είναι ο λόγος που οι συναρτήσεις area, perimeter και diagonal δεν λαμβάνουν παραμέτρους όπως οι αντίστοιχες στατικές συναρτήσεις rectArea, rectPerimeter και rectDiagonal.

Παρόλα αυτά, ο κώδικάς μας έτσι όπως έχει, συνεχίζει να περιλαμβάνει σημαντικές αδυναμίες. Προσέξτε πως η εφαρμογή δίνει τιμές στις διαστάσεις ενός Rectangle. Τι θα γίνει αν για παράδειγμα δοθεί αρνητική τιμή σε κάποια από τις διαστάσεις, π.χ. length=2, width=-2; Θα έχουμε δημιουργήσει ένα ορθογώνιο που δεν θα είναι σωστό. Στη συνέχεια, αν ζητήσουμε το εμβαδόν του, θα λάβουμε την αρνητική τιμή -4 που είναι χωρίς νόημα και αν ζητήσουμε την περίμετρό του θα λάβουμε την τιμή 0, επίσης λανθασμένη. Χρησιμοποιώντας τον προκαθορισμένο δημιουργό έχουμε μεταθέσει στην εφαρμογή την ευθύνη να δώσει τιμές στα length και width. Η δημιουργία όμως αντικειμένων της κλάσης γίνεται πιο ασφαλής αν ολοκληρώνεται από την ίδια την κλάση. Η κλάση γνωρίζει τους περιορισμούς στους οποίους υπόκεινται τα μέλη της, ενώ η εφαρμογή δεν είναι αναγκαίο να τους γνωρίζει.

11.1 Οι δημιουργοί

Ο έλεγχος της δημιουργίας των αντικειμένων από την κλάση γίνεται με τη βοήθεια των δημιουργών (constructors) της κλάσης. Ένας δημιουργός είναι μια συνάρτηση μέλος της κλάσης που έχει όνομα ίδιο με της κλάσης και δεν έχει τιμή επιστροφής. Μέσα στον δημιουργό, η κλάση μπορεί να εκτελέσει τους απαραίτητους ελέγχους ώστε να διασφαλισθεί ότι τα στιγμιότυπα που δημιουργούνται είναι σωστά. Στον κώδικα 11.5 παρουσιάζουμε έναν δημιουργό για την κλάση Rectangle.

```
public Rectangle(double len, double wid) {
    if (len <= 0 || wid <= 0) {
        throw new IllegalArgumentException();
    }
    length = len;
```

```
        width = wid;
    }
```

Κώδικας 11.5 *Δημιουργός της κλάσης Rectangle*

Αυτός ο δημιουργός λαμβάνει δύο παραμέτρους, η μια αντιπροσωπεύει το μήκος και η άλλη το πλάτος του ορθογώνιου που δημιουργείται κατά την κλήση του. Ο δημιουργός ελέγχει πρώτα αν οι τιμές των παραμέτρων είναι αποδεκτές. Αν όχι, προχωράει στη διαχείριση του λάθους. Στη δική μας περίπτωση η διαχείριση αυτή συνίσταται στην παραγωγή κατάλληλης εξαίρεσης που τερματίζει την εφαρμογή παράγοντας αντίστοιχο μήνυμα. Αν όμως οι παράμετροι περάσουν τον έλεγχο επιτυχώς, τότε οι τιμές τους αντιγράφονται στις μεταβλητές στιγμιότυπου από τον κώδικα που ακολουθεί και συντελείται η δημιουργία ενός ορθού αντικειμένου.

Η κλήση του νέου δημιουργού γίνεται κατά τη δημιουργία των στιγμιότυπων με κώδικα της μορφής:

```
Rectangle r = new Rectangle(2, 5);
Rectangle t = new Rectangle(3, 3);
```

Εδώ δημιουργούμε δύο ορθογώνια, το ένα έχει μήκος 2 και πλάτος 5 και το άλλο είναι τετράγωνο με μήκος πλευράς 3. Ιδιαίτερη προσοχή θα πρέπει να δοθεί στο γεγονός ότι η πρόσθεση ενός δημιουργού στην κλάση ακυρώνει αυτόματα τον προκαθορισμένο δημιουργό. Με άλλα λόγια ο κώδικας 11.4 δεν μεταγλωττίζεται πλέον. Ο προκαθορισμένος δημιουργός είναι αναγκαστικά δημιουργός χωρίς παραμέτρους. Αν επιθυμούμε ένα δημιουργό χωρίς παραμέτρους, τότε θα πρέπει να ορίσουμε σαφώς έναν τέτοιο μέσα στην κλάση. Για παράδειγμα:

```
public Rectangle() {
     length = 2;
     width = 2;
}
```

Ο κώδικας 11.4 μεταγλωττίζεται τώρα. Ωστόσο, το ορθογώνιο που δημιουργείται δεν έχει μήκος και πλάτος 0 αλλά 2. Γενικά μπορούμε να προσθέσουμε όσους δημιουργούς απαιτούνται σε μια κλάση αρκεί να ακολουθούν τους κανόνες υπερφόρτωσης συναρτήσεων (function overloading), δηλαδή να διαφοροποιείται η ταυτότητά τους (signature). Για παράδειγμα, μπορούμε να προσθέσουμε έναν δημιουργό για να κατασκευάζουμε τετράγωνα. Δεδομένου ότι το μήκος ενός τετραγώνου είναι ίσο με το πλάτος του, ένας τέτοιος δημιουργός θα χρειάζεται μια παράμετρο.

```
public Rectangle(double len) {
     if (len <= 0) {
          throw new IllegalArgumentException();
     }
     length = len;
     width = len;
}
```

Εφόσον προσθέσουμε έναν τέτοιο δημιουργό στην κλάση, η κλήση Rectangle r=new Rectangle(5), αναφέρεται στον δημιουργό που έχει μια παράμετρο και κατασκευάζει ένα τετράγωνο με πλευρά μήκους 5.

11.2 Αναγνώστες και Ρυθμιστές

Η νέα μορφή της Rectangle μας εξασφαλίζει πως τα αντικείμενα, εφόσον δημιουργούνται, είναι ορθά. Παρόλα αυτά, ο κώδικάς μας έχει πάλι σοβαρές αδυναμίες. Παρότι έχουμε εγγυηθεί την ορθή δημιουργία των αντικειμένων, δεν τα προστατεύουμε καθόλου μετά τη δημιουργία τους. Η εφαρμογή μπορεί να προσπελάσει τα ευαίσθητα μέλη της κλάσης και να εκχωρήσει λανθασμένες τιμές μέσω του τελεστή τελεία (dot operator). Ένας κώδικας της μορφής

```
Rectangle r = new Rectangle(2, 5);
```

```
r.length=-5;
```

καταστρέφει την ακεραιότητα του αντικειμένου r.

Λύση στο πρόβλημα μας δίνει η δήλωση ως private των μεταβλητών length και width, οπότε ο κώδικας πελάτη δεν μπορεί να τις προσπελάσει. Με άλλα λόγια, κώδικας της μορφής r.length=-5 έξω από την κλάση δεν μεταγλωττίζεται εφόσον η length έχει χαρακτηριστεί private. Επομένως, έχουμε εγγυηθεί πως το αντικείμενο είναι ορθό κατά τη δημιουργία του και δεν μπορεί να καταστρέψει την ορθότητά του ο χρήστης της κλάσης. Ωστόσο, μπορεί να θέλουμε να παραχωρήσουμε πρόσβαση στον χρήστη ώστε να μπορεί να διαβάσει τις τιμές ενός private μέλους. Τον ρόλο αυτό έχουν αναλάβει ειδικές συναρτήσεις της κλάσης γνωστές ως αναγνώστες (getters). Επίσης, είναι πιθανό να θέλουμε να επιτρέψουμε στον κώδικα πελάτη να γράψει ένα private μέλος υπό τον έλεγχο της κλάσης. Τον ρόλο αυτό αναλαμβάνουν οι ρυθμιστές (setters) της κλάσης. Στη συνέχεια παρουσιάζεται η κλάση Rectangle με private μεταβλητές στιγμιότυπου, getters και setters.

```
public class Rectangle {

    private double length, width;

    public Rectangle(double len, double wid) {
        if (len <= 0 || wid <= 0) {
            throw new IllegalArgumentException();
        }
        length = len;
        width = wid;
    }

    public Rectangle(double len) {
        if (len <= 0) {
            throw new IllegalArgumentException();
        }
        length = len;
        width = len;
    }

    public Rectangle() {
        length = 2;
        width = 2;
    }

    public double area() {
        return length * width;
    }

    public double perimeter() {
        return 2 * (length + width);
    }

    public double diagonal() {
        return Math.sqrt(Math.pow(length, 2) + Math.pow(width, 2));
    }

    public double getLength() {
        return length;
    }

    public void setLength(double len) {
        if (len <= 0) {
            throw new IllegalArgumentException();
        }
```

```
            length = len;
        }

        public double getWidth() {
            return width;
        }

        public void setWidth(double wid) {
            if (wid <= 0) {
                throw new IllegalArgumentException();
            }
            width = wid;
        }
    }
```

Κώδικας 11.6 *Η κλάση Rectangle με ιδιωτικές μεταβλητές, getters και setters*

Ο κώδικας 11.7 παρουσιάζει μια εφαρμογή που αξιοποιεί την κλάση Rectangle όπως έχει διαμορφωθεί μέχρι στιγμής.

```
public class App {

    public static void main(String[] args) {
        Rectangle r = new Rectangle(2, 5);
        Rectangle t = new Rectangle(3);
        r.setLength(5);
        System.out.println(t.getLength()+" "+t.getWidth());
        System.out.println(r.area());
        System.out.println(r.perimeter());
        System.out.println(r.diagonal());
      }
}
```

Κώδικας 11.7 *Εφαρμογή που αξιοποιεί την κλάση Rectangle με ασφάλεια*

Όπως βλέπουμε, ο χρήστης της κλάσης μπορεί να δημιουργήσει αντικείμενα Rectangle χρησιμοποιώντας έναν από τους διαθέσιμους δημιουργούς, να διαβάσει και να γράψει με ασφάλεια το μήκος και το πλάτος ενός τέτοιου αντικειμένου και να υπολογίσει το εμβαδόν, την περίμετρο και το μήκος της διαγωνίου χωρίς να αντιμετωπίσει κανένα από τα προβλήματα του διαδικαστικού μοντέλου όπως αυτά περιεγράφηκαν προηγουμένως σε αυτήν την ενότητα.

Ας σημειωθεί εδώ πως οι αναγνώστες ονομάζονται με το πρόθεμα get ακολουθούμενο από το όνομα του μέλους στο οποίο αναφέρονται. Σχηματίζεται έτσι σύνθετη λέξη και ισχύει η σύμβαση πως στα αναγνωριστικά (identifiers) που συνιστούν σύνθετες λέξεις, κάθε συνθετικό πλην του πρώτου θα πρέπει να αρχίζει από κεφαλαίο, π.χ. getLenth(). Αντίστοιχες συμβάσεις ισχύουν και για τους ρυθμιστές, μόνο που εδώ το πρόθεμα get αντικαθίσταται από το πρόθεμα set. Εξαίρεση σε αυτόν τον κανόνα αποτελούν οι αναγνώστες των μεταβλητών boolean στα ονόματα των οποίων αντί get έχουμε το πρόθεμα is.

11.3 Προσδιοριστές Προσπέλασης

Όπως βλέπουμε στον κώδικα 11.6, έχουμε προσδιορίσει την κλάση ως public. Αυτό σημαίνει ότι μπορεί να χρησιμοποιηθεί από οποιονδήποτε, δηλαδή από οποιαδήποτε κλάση στο ίδιο ή σε άλλο πακέτο και φυσικά από την ίδια την κλάση Rectangle. Επίσης ως public έχουν προσδιοριστεί οι δημιουργοί και τα υπόλοιπα μέλη-συναρτήσεις της κλάσης. Επομένως, επιτρέπεται η προσπέλασή τους από κάθε χρήστη της κλάσης είτε βρίσκεται στο ίδιο ή σε άλλο πακέτο. Η πρόσβαση που καθορίζεται από την απουσία προσδιοριστή προσπέλασης, ονομάζεται προκαθορισμένη προσπέλαση ή προσπέλαση πακέτου (default access ή package access). Σύμφωνα με αυτήν, η πρόσβαση στα μέλη της κλάσης επιτρέπεται μόνο από κλάσεις που ορίζονται στο ίδιο πακέτο. Έτσι αν ορίσουμε τον ρυθμιστή του πλάτους ως

```
    void setWidth(double wid) {
```

```
        if (wid<=0)
              throw new IllegalArgumentException();
        width = wid;
    }
```

τότε μόνο οι κλάσεις που ανήκουν στο ίδιο πακέτο με την Rectangle θα μπορούν να τον καλέσουν και άρα μόνο αυτές θα μπορούν να μεταβάλουν το πλάτος ενός ορθογώνιου μετά τη δημιουργία του.
Τα μέλη-δεδομένα έχουν χαρακτηριστεί ως private. Αυτό σημαίνει πως επιτρέπεται η προσπέλασή τους μόνο μέσα στην κλάση Rectangle. Αν ωστόσο κάποιος χρήστης της κλάσης επιθυμεί να διαβάσει ή να γράψει τα μέλη αυτά μπορεί να χρησιμοποιήσει τους δημόσιους ρυθμιστές και αναγνώστες.
Τέλος, υπάρχει και ο προσδιοριστής protected που σχετίζεται με την κληρονομικότητα και παρουσιάζεται στην ενότητα 13.1.2.

11.4 Η λέξη κλειδί this

Προσέξτε πως οι μη στατικές συναρτήσεις καλούνται αναγκαστικά από ένα στιγμιότυπο της κλάσης. Μερικές φορές μέσα στον ορισμό μιας μη στατικής συνάρτησης είναι δυνατόν να χρειάζεται να αναφερθούμε στο στιγμιότυπο που την καλεί. Βεβαίως η συνάρτηση μπορεί να κληθεί από πολλά διαφορετικά στιγμιότυπα, κανένα εκ των οποίων δεν είναι γνωστό κατά τον χρόνο που αναπτύσσουμε τη συνάρτηση. Για τον λόγο αυτόν, η Java περνάει μια αναφορά στο στιγμιότυπο που κάλεσε τη συνάρτηση μέσω της λέξης-κλειδί this.

Σημειώστε πως επειδή οι στατικές συναρτήσεις δεν καλούνται υποχρεωτικά από αντικείμενο της κλάσης, δεν επιτρέπεται η χρήση του this μέσα σε στατική συνάρτηση.

Ακολουθεί παράδειγμα χρήσης της λέξης-κλειδί this.

```
public void setLength(double length) {
     if (length<=0)
          throw new IllegalArgumentException();
     this.length = length;
}
```

Η παράμετρος στη συνάρτηση setLength έχει ίδιο όνομα με τη μεταβλητή στιγμιότυπου length. Η παράμετρος λειτουργεί σαν τοπική μεταβλητή και αποκρύπτει τη μεταβλητή length του στιγμιότυπου. Το αναγνωριστικό length μέσα στην setLength αναφέρεται στην παράμετρο. Αν θέλουμε μέσα στην setLength να αναφερθούμε στη μεταβλητή του στιγμιότυπου μπορούμε να χρησιμοποιήσουμε το this όπως φαίνεται στον παραπάνω κώδικα.

Ανάλογα με τα συμφραζόμενα (context), η λέξη-κλειδί this, μπορεί επιπλέον να χρησιμοποιηθεί για την κλήση ενός δημιουργού της κλάσης μέσα σε έναν άλλο δημιουργό.

```
public Rectangle(double len) {
     this(length,length);
}
```

Εδώ ξαναγράφουμε τον δημιουργό κατασκευής τετραγώνων. Το τετράγωνο δημιουργείται με κλήση στον δημιουργό που έχει δύο παραμέτρους και επιτυγχάνεται με χρήση του this. Ας σημειωθεί πως τέτοια κλήση μπορεί να γίνει μόνο στην πρώτη γραμμή ενός δημιουργού.

11.5 Εμβέλεια και χρόνος ζωής

Τα στατικά μέλη της κλάσης φορτώνονται στη μνήμη συνήθως όταν η εφαρμογή χρειασθεί την κλάση στην οποία ανήκουν. Στη συνέχεια διατηρούνται μέχρι το τέλος της εφαρμογής. Επομένως, αυτά τα όρια διαμορφώνουν τον χρόνο ζωής τους. Αντίθετα, τα μέλη στιγμιότυπου δημιουργούνται ταυτόχρονα με το στιγμιότυπο στο οποίο ανήκουν και καταστρέφονται μαζί με αυτό.

Η εμβέλεια των μελών της κλάσης καθορίζεται από τον προσδιοριστή προσπέλασης του μέλους σε συνδυασμό με τον προσδιοριστή προσπέλασης της κλάσης. Για παράδειγμα, ένα μέλος που ανήκει σε δημόσια κλάση και είναι δημόσιο και το ίδιο μπορεί να προσπελαστεί από οποιονδήποτε χρήστη της κλάσης.

Ένα μέλος όμως που ανήκει σε δημόσια κλάση αλλά είναι το ίδιο ιδιωτικό μπορεί να προσπελαστεί μόνο από την κλάση στην οποία ανήκει. Παρόμοια, ένα μέλος που είναι δημόσιο αλλά ανήκει σε κλάση με πρόσβαση πακέτου μπορεί να προσπελαστεί μόνο από κλάσεις του ίδιου πακέτου.

11.6 Άλλες αναγκαίες συναρτήσεις της κλάσης

Προκειμένου η κλάση μας να είναι ολοκληρωμένη θα πρέπει να διαθέτει ως μέλη τρεις ακόμη συναρτήσεις, την toString, την equals και την hashCode.

11.6.1 Η συνάρτηση toString

Αν προσπαθήσουμε να εκτυπώσουμε ένα αντικείμενο της κλάσης Rectangle, θα λάβουμε ως έξοδο μια περίεργη εκ πρώτης όψεως αλφαριθμητική σειρά. Για παράδειγμα, ο ακόλουθος κώδικας

```
Rectangle k=new Rectangle(3, 6);
System.out.println(k);
```

θα τυπώσει κάτι σαν PackageName.Rectangle@4769b07b.
Πρόκειται για το όνομα της κλάσης, ακολουθούμενο από τον κωδικό κατακερματισμού του στιγμιότυπου k εκφρασμένο σε δεκαεξαδική μορφή. Στις περισσότερες περιπτώσεις όμως που επιχειρούμε να τυπώσουμε ένα στιγμιότυπο, εκείνο που θέλουμε είναι η τιμή του στιγμιότυπου και όχι το όνομα της κλάσης συνοδευόμενο από τον κωδικό κατακερματισμού. Η Java μας δίνει τη δυνατότητα να το πετύχουμε ορίζοντας κατάλληλα τη συνάρτηση toString().

```
@Override
public String toString() {
     return "Rectangle{" + "length=" + length + ", width=" +
     width + '}';
}
```

Η συνάρτηση toString είναι υποχρεωτικά public και επιστρέφει μια αναπαράσταση του στιγμιότυπου που την καλεί με τη μορφή αλφαριθμητικής σειράς. Επιπλέον, από τον ορισμό της συνάρτησης προηγείται η ειδική σημείωση (annotation) @Override. Η σημείωση αυτή δεν είναι υποχρεωτική από τον μεταγλωττιστή, ωστόσο είναι χρήσιμη και δεν θα πρέπει να ξεχνάμε να την προσθέτουμε. Το νόημά της σχετίζεται με την κληρονομικότητα και το παρουσιάζουμε στην ενότητα 13.1.1. Στο σώμα της συνάρτησης toString βάζουμε τον κώδικα που καθορίζει τι θα παίρνουμε σε όλες τις περιπτώσεις που χρησιμοποιούμε ένα στιγμιότυπο της κλάσης αλλά χρειάζεται μια αλφαριθμητική αναπαράστασή του. Έτσι τώρα, η εκτύπωση του αντικειμένου k θα τυπώσει Rectangle{length=3.0, width=6.0}. Η συνάρτηση toString() καλείται αυτόματα (implicitly) όπου είναι απαραίτητο, αλλά μπορεί να κληθεί και σαφώς (explicitly) από την εφαρμογή.

11.6.2 Η συνάρτηση equals

Μια άλλη αναγκαία συνάρτηση είναι η equals. Όπως έχουμε αναφέρει, τα στιγμιότυπα συνιστούν μεταβλητές αναφοράς. Επομένως, η σύγκριση μεταξύ δύο στιγμιότυπων με τον τελεστή == (equality operator) συγκρίνει τις διευθύνσεις τους και όχι τις τιμές τους. Έστω ο κώδικας

```
Rectangle r1=new Rectangle(2,6), r2=new Rectangle(2,6);
System.out.println(r1==r2);
```

Παρόλο που r1 και r2 είναι 2 ορθογώνια με ίδιες διαστάσεις, ο έλεγχος ισότητας εδώ μας επιστρέφει false. Πράγματι, έχουμε 2 διαφορετικά στιγμιότυπα. Η μεταβλητή r1 αναφέρεται στη μνήμη που έχει κατανεμηθεί για το ένα ορθογώνιο και η r2 στη μνήμη του άλλου. Επομένως, το αποτέλεσμα false είναι αναμενόμενο. Πώς όμως μπορούμε να συγκρίνουμε τα ίδια τα στιγμιότυπα και όχι τις διευθύνσεις τους; Εδώ έρχεται η συνάρτηση equals να δώσει τη λύση.

```
@Override
public boolean equals(Object obj) {
     if (this == obj) {
          return true;
     }
     if (obj == null) {
          return false;
     }
     if (getClass() != obj.getClass()) {
          return false;
     }
     final Rectangle other = (Rectangle) obj;
     if (Double.compare(width, other.width) != 0) {
          return false;
     }
     return Double.compare(length, other.length) == 0;
}
```

Η equals είναι αναγκαστικά public, επιστρέφει boolean και διαθέτει μια τυπική παράμετρο τύπου Object. Επιπλέον, η equals πρέπει να πληροί τις ακόλουθες ιδιότητες [1]:

1. Ανακλαστική: για οποιαδήποτε μη-μηδενική τιμή αναφοράς x, η έκφραση x.equals (x) πρέπει να επιστρέφει true.
2. Συμμετρική: για οποιεσδήποτε μη μηδενικές τιμές αναφοράς x και y, x.equals (y) πρέπει να επιστρέφει true εάν και μόνο εάν το y.equals (x) επιστρέφει true.
3. Μεταβατική: για οποιεσδήποτε μη μηδενικές τιμές αναφοράς x, y και z, εάν x.equals (y) επιστρέφει true και y.equals (z) επιστρέφει true, τότε x.equals (z) θα πρέπει να επιστρέψει true.
4. Για οποιεσδήποτε μη-μηδενικές τιμές αναφοράς x και y, όλες οι κλήσεις του x.equals (y) επιστρέφουν σταθερά true ή σταθερά false, υπό την προϋπόθεση ότι x και y παραμένουν σταθερά.
5. Για οποιαδήποτε μη-μηδενική τιμή αναφοράς x, x.equals (null) θα πρέπει να επιστρέφει false.

Μέσα στην equals αρχικά γίνεται έλεγχος αν το στιγμιότυπο που κάλεσε την equals είναι το ίδιο με την παράμετρό της. Σε αυτήν την περίπτωση επιστρέφεται true. Στη συνέχεια ελέγχεται αν η παράμετρος είναι null. Αν είναι, με δεδομένο ότι το στιγμιότυπο που κάλεσε την equals δεν είναι null, επιστρέφεται false. Μετά ελέγχεται αν η παράμετρος ανήκει στην ίδια κλάση με το στιγμιότυπο που κάλεσε την equals. Αν τα δύο αντικείμενα ανήκουν σε διαφορετικές κλάσεις, επιστρέφεται false. Στο επόμενο βήμα ελέγχονται οι μεταβλητές στιγμιότυπου των δύο αντικειμένων μια προς μια για ισότητα. Αν κάποιος από τους ελέγχους ισότητας αποτύχει, επιστρέφεται false. Τέλος, εφόσον η equals δεν έχει ήδη επιστρέψει, τα αντικείμενα θεωρούνται ίσα και επιστρέφεται true. Γενικά, μέσω της equals δίνουμε τον ορισμό ισότητας μεταξύ των αντικειμένων της κλάσης.

Όπως γνωρίζουμε, ο έλεγχος της ισότητας μεταξύ πραγματικών γίνεται συνήθως προσεγγιστικά, δηλαδή αν η απόλυτη τιμή της διαφοράς δύο πραγματικών είναι μικρότερη από μία οριακή τιμή που συνήθως ονομάζεται epsilon, οι αριθμοί θεωρούνται ίσοι. Ωστόσο, μια τέτοια προσεγγιστική ισότητα δεν πληροί τη μεταβατική ιδιότητα, επομένως η συνάρτηση equals δεν μπορεί να βασιστεί σε προσεγγιστική ισότητα μεταξύ των μελών της. Αν χρειάζεστε μια προσεγγιστική ισότητα, εφοδιάστε την κλάση σας με κατάλληλη συνάρτηση.

```
public boolean aproximateEquals(Rectangle r) {
     if (equals(r)) {
          return true;
     }
     if (Math.abs(width - r.width) < 0.001 && Math.abs(length
     - r.length) < 0.001) {
          return true;
     }
```

```
        return false;
    }
```

Τέλος, ας σημειωθεί ότι ο τύπος της παραμέτρου της equals είναι Object. Ωστόσο, η πραγματική παράμετρος μπορεί να είναι Rectangle. Πρόκειται επίσης για ένα θέμα που σχετίζεται με την κληρονομικότητα και εξηγείται στην ενότητα 13.

11.6.3 Η συνάρτηση hashCode

Τα αντικείμενα της κλάσης μας ενδέχεται να φορτωθούν σε συλλογές κατακερματισμού (hash Collections), όπως το HashMap ή το HashSet. Οι συλλογές αυτές χρειάζονται ένα κλειδί κατακερματισμού (hash-key) για κάθε αντικείμενο που αποθηκεύουν το οποίο χρησιμοποιούν ώστε να επιτυγχάνουν ταχύτερη προσπέλαση. Καθώς κάθε αντικείμενο είναι πιθανό κάποια στιγμή να αποθηκευτεί σε συλλογή κατακερματισμού, θα πρέπει κάθε αντικείμενο να είναι σε θέση να υπολογίζει την τιμή κατακερματισμού του. Αυτός ακριβώς είναι ο σκοπός της συνάρτησης hashCode, να επιστρέψει μια τιμή κατακερματισμού για το αντικείμενο που την καλεί.

Η ανάπτυξη της hashCode πρέπει υποχρεωτικά να υπακούει σε τρεις κανόνες:

1. Κάθε κλήση της που γίνεται κατά τη διάρκεια της εκτέλεσης μιας εφαρμογής θα πρέπει να επιστρέφει το ίδιο hashCode για το ίδιο αντικείμενο. Μεταξύ διαφορετικών εκτελέσεων της ίδιας εφαρμογής δεν είναι απαραίτητο να επιστρέφεται ο ίδιος κωδικός κατακερματισμού.
2. Δύο αντικείμενα για τα οποία η equals επιστρέφει true θα πρέπει να έχουν τον ίδιο κωδικό κατακερματισμού
3. Δύο αντικείμενα για τα οποία η equals επιστρέφει false δεν είναι υποχρεωτικό να επιστρέφουν διαφορετικό κωδικό κατακερματισμού. Είναι όμως χρήσιμο. Στην περίπτωση για παράδειγμα που μια συνάρτηση hashCode επιστρέφει 1 για κάθε αντικείμενο μιας κλάσης, τότε η προσπέλαση μέσα στις συλλογές κατακερματισμού θα μετατραπεί σε σειριακή.

Γενικότερα, μία καλή συνάρτηση κατακερματισμού έχει ως στόχο την ελαχιστοποίηση παραγωγής ίδιων κωδικών για διαφορετικά αντικείμενα.
Ακολουθεί μια απλή συνάρτηση κατακερματισμού για την κλάση Rectangle.

```
@Override
public int hashCode() {
    return Double.hashCode(length) + Double.hashCode(width);
}
```

Γενικά μια συνάρτηση hashCode επιστρέφει έναν int, η τιμή του οποίου είναι συνάρτηση των τιμών των μεταβλητών του αντικειμένου που καλεί την hashCode. Εφόσον, οι μεταβλητές του αντικειμένου είναι αντικείμενα και οι ίδιες, μπορούμε να αξιοποιήσουμε τη συνάρτηση hashCode της κλάσης που ανήκει η μεταβλητή. Στην περίπτωσή που οι μεταβλητές είναι θεμελιώδεις (native) τύποι μπορούμε να χρησιμοποιήσουμε τη στατική hashCode της αντίστοιχης κλάσης όπως κάναμε σε αυτό το παράδειγμα.

Όπως αναφέραμε προηγουμένως, η συνάρτηση κατακερματισμού που παρουσιάζουμε είναι αρκετά απλή. Πληροί μεν τους κανόνες της συνάρτησης κατακερματισμού αλλά δεν είναι ιδιαίτερα αποτελεσματική. Για παράδειγμα, δύο αντικείμενα Rectangle, το ένα με μήκος 2 και πλάτος 6 και το άλλο με μήκος 6 και πλάτος 2, θα παράξουν τον ίδιο κωδικό κατακερματισμού παρότι είναι διαφορετικά, δηλαδή η equals επιστρέφει false.

Αυτό βέβαια το πρόβλημα μπορεί να αντιμετωπιστεί αν πολλαπλασιάσουμε τη μία διάσταση με έναν ακέραιο. Πάντως, όλα τα σύγχρονα IDE παράγουν αυτόματα τη συνάρτηση hashCode. Για παράδειγμα, το NetBeans, για την κλάση Rectangle, παραγάγει την ακόλουθη συνάρτηση κατακερματισμού:

```
@Override
public int hashCode() {
    int hash = 3;
```

```
        hash       =       53       *       hash       +       (int)
        (Double.doubleToLongBits(this.length)                      ^
        (Double.doubleToLongBits(this.length) >>> 32));
        hash       =       53       *       hash       +       (int)
        (Double.doubleToLongBits(this.width)                       ^
        (Double.doubleToLongBits(this.width) >>> 32));
        return hash;
    }
```

Η doubleToLongBits(length) επιστρέφει έναν long που όταν η δυαδική του αναπαράσταση μετατραπεί σε double, το αποτέλεσμα θα είναι ακριβώς ίσο με length. Στο αποτέλεσμα αυτό γίνεται μη προσημασμένη δεξιά διολίσθηση 32 θέσεων και αυτό που επιστρέφεται γίνεται αποκλειστικό or πάλι με την doubleToLongBits(length). Το νέο αποτέλεσμα το μετατρέπουμε σε int και προσθέτουμε έναν ακέραιο. Οι σταθερές 3 και 53, πιθανότατα να μην είναι ίδιες σε επαναλαμβανόμενες αιτήσεις προς το NetBeans να μας φτιάξει τη συνάρτηση κατακερματισμού της κλάσης Rectangle.

Περισσότερες πληροφορίες για το πώς μπορείτε να δημιουργείτε αποτελεσματικότερες συναρτήσεις κατακερματισμού μπορείτε να βρείτε στα βιβλία των Adekunle [2] και Aumasson, et all. [3].
Ακολουθεί η τελική έκδοση της κλάσης Rectangle στον κώδικα 11.8.

```
public class Rectangle {

    private double length, width;

    public Rectangle(double length, double width) {
        if (length <= 0 || width <= 0) {
            throw new IllegalArgumentException();
        }
        this.length = length;
        this.width = width;
    }

    public Rectangle(double length) {
        this(length, length);
    }

    public Rectangle() {
        this.length = 2;
        this.width = 2;
    }

    public double area() {
        return length * width;
    }

    public double perimeter() {
        return 2 * (this.length + this.width);
    }

    public double diagonal() {
        return Math.sqrt(Math.pow(this.length, 2) + Math.pow(this.width,
2));
    }

    public double getLength() {
        return this.length;
    }

    public void setLength(double length) {
        if (length <= 0) {
```

```
            throw new IllegalArgumentException();
        }
        this.length = length;
    }

    public double getWidth() {
        return this.width;
    }

    public void setWidth(double width) {
        if (width <= 0) {
            throw new IllegalArgumentException();
        }
        this.width = width;
    }

    @Override
    public String toString() {
        return "Rectangle{" + "length=" + length + ", width=" + width + '}';
    }

    @Override
    public int hashCode() {
        return Double.hashCode(length) + Double.hashCode(width);
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Rectangle other = (Rectangle) obj;
        if (Double.compare(width, other.width) != 0) {
            return false;
        }
        return Double.compare(length, other.length) == 0;
    }

    public boolean aproximateEquals(Rectangle r) {
        if (equals(r)) {
            return true;
        }
        return Math.abs(width - r.width) < 0.001 && Math.abs(length -
r.length) < 0.001;
    }
}
```

Κώδικας 11.8 *Υπόδειγμα ολοκληρωμένης κλάσης*

11.7 Απαριθμήσιμοι τύποι

Μερικές φορές στα προγράμματά μας χρειαζόμαστε ένα ή περισσότερα σύνολα από σταθερές. Για παράδειγμα, έστω ότι θέλουμε να αναπαραστήσουμε με σταθερές κάθε ημέρα της εβδομάδας έτσι ώστε να μην αναφερόμαστε σε κυριολεκτικούς ακέραιους που κάνουν το πρόγραμμά μας δυσανάγνωστο.
Για την καλύτερη ικανοποίηση τέτοιων αναγκών, η Java προσφέρει τον τύπο enum. Στον κώδικα 11.9, παρουσιάζουμε τη δήλωση ενός απαριθμήσιμου τύπου ορισμού των ημερών της εβδομάδας.

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

Κώδικας 11.9 *Απαριθμήσιμος τύπος για τις ημέρες της εβδομάδας*

Όπως φαίνεται στον κώδικα 11.9, η δήλωση απαριθμήσιμου τύπου γίνεται με χρήση της λέξης-κλειδί enum. Κάθε μια από τις μεταβλητές MONDAY έως FRIDAY είναι και μία σταθερά. Έτσι, τα αντίστοιχα αναγνωριστικά εκφράζονται με κεφαλαία ώστε να πληρούνται οι συμβάσεις ονοματολογίας των σταθερών. Επιπλέον, οι μεταβλητές αυτές είναι εξ ορισμού public και static. Από τη στιγμή που έχουμε διαθέσιμο τον τύπο Day, μπορούμε να τον χρησιμοποιήσουμε στον κώδικά μας, καταρχάς δηλώνοντας μεταβλητές τύπου Day, π.χ.

```
Day d=Day.FRIDAY;
```

Στη συνέχεια μπορούμε να χρησιμοποιήσουμε κανονικά τη μεταβλητή d, όπως για παράδειγμα δείχνει ο κώδικας που ακολουθεί:

```
switch (d) {
     case MONDAY:
          System.out.println("First day of the Week");
          break;
     case THUSDAY:
          System.out.println("Second day of the Week");
          break;
     case WEDNESDAY:
          System.out.println("It's a long Week");
          break;
     case THURSDAY:
          System.out.println("I fill tired");
          break;
     case FRIDAY:
          System.out.println("Last day of the Working Week");
          break;
     case SATURDAY:
          System.out.println("It's a terrific day");
          break;
     case SUNDAY:
          System.out.println("Rest today");
          break;
}
```

Ας δούμε όμως μερικά μυστικά αυτού του τύπου. Καταρχάς, πρόκειται για εξειδικευμένη κλάση. Οι δε τιμές MONDAY έως FRIDAY της Day δεν είναι παρά αντικείμενα της Day που έχουν δημιουργηθεί με τον default δημιουργό. Βεβαίως, όπως θα δούμε αμέσως παρακάτω, μπορούμε να ορίσουμε και δικούς μας δημιουργούς. Επιπλέον, τα αντικείμενα αυτά είναι public, static και final παρότι δεν έχουν δηλωθεί σαφώς ως τέτοια. Ο τύπος ονομάζεται απαριθμήσιμος καθώς τα αντικείμενα-τιμές αντιστοιχίζονται σε μία ακέραιη τιμή. Οι εξ ορισμού τιμές είναι 0,1,2,..,ν, για ν+1 τιμές ενός απαριθμήσιμου τύπου. Έτσι η τιμή της MONDAY είναι 0, της THUSDAY 1, κοκ, έως την SUNDAY με τιμή 6. Ο τύπος μάλιστα διαθέτει μια μέθοδο, την ordinal, που μας επιστρέφει την τιμή ενός απαριθμήσιμου αντικειμένου, π.χ. ο κώδικας

```
Day d = Day.FRIDAY;
System.out.println(d.ordinal());
```

θα τυπώσει 4.
Επιπλέον, η μέθοδος toString θα επιστρέψει σε μορφή String την ονομασία του αντικειμένου, δηλαδή ο κώδικας

```
Day d = Day.FRIDAY;
System.out.println(d);
```

εμφανίζει FRIDAY.
Επίσης, ο τύπος διαθέτει τη στατική μέθοδο values που επιστρέφει τα αντικείμενα σε πίνακα, δηλαδή η πρόταση

```
System.out.println(Arrays.asList(Day.values()));
```

θα εμφανίσει

```
[MONDAY, THUSDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY]
```

Τέλος, να επισημάνουμε πως ο εξ ορισμού δημιουργός του τύπου enum είναι ιδιωτικός ώστε να μην έχει ο χρήστης του την δυνατότητα να δημιουργήσει νέα αντικείμενα και να αλλοιώσει τον χαρακτήρα του τύπου.
Ας δούμε όμως και ένα άλλο παράδειγμα, όπου η αρίθμηση δεν ακολουθεί την τυπική περίπτωση από 0..ν. Στον κώδικα 11.10 δίνουμε έναν απαριθμήσιμο τύπο που αντιπροσωπεύει τους βαθμούς των καρτών μιας τράπουλας, έτσι ώστε το 2 της τράπουλας να έχει αρίθμηση 2, το 3 να έχει αρίθμηση 3 και το μοτίβο αυτό να ακολουθείται έως το 10, ενώ ο Βαλές έχει αρίθμηση 11, η Ντάμα 12, ο Ρήγας 13 και ο Άσος 14.

```
import java.util.Arrays;

public enum CardRank {

    TWO(2),
    THREE(3),
    FOUR(4),
    FIVE(5),
    SIX(6),
    SEVEN(7),
    EIGHT(8),
    NINE(9),
    TEN(10),
    JACK(11),
    QUEEN(12),
    KING(13),
    ACE(14);

    private final int cardValue;

    private CardRank(int value) {
        this.cardValue = value;
    }

    public int getCardValue() {
        return cardValue;
    }
}
```

Κώδικας 11.10 *Ο απαριθμήσιμος τύπος CardRank*

Καθώς δεν μας διευκολύνει η εξ ορισμού αρίθμηση, έχουμε φτιάξει έναν δημιουργό με τον οποίο δίνουμε τιμή στην cardValue την οποία μπορούμε να χρησιμοποιούμε σαν τιμή των μεταβλητών του τύπου. Κατά τη δημιουργία των αντικειμένων TWO έως ACE καλείται τώρα ο νέος δημιουργός και όπου παραστεί ανάγκη μπορούμε να πάρουμε ως αρίθμηση των αντικειμένων την cardValue μέσω της getCardValue. Προσοχή όμως, η τιμή αυτή δεν αντικαθιστά την τιμή που επιστρέφει η ordinal. Έτσι, η μεν CardRank.NINE.getCardValue() θα επιστρέψει 9, η δε CardRank.NINE.ordinal() θα επιστρέψει 7.

Ένα άλλο σημαντικό στοιχείο στον κώδικα 11.10 είναι πως ο δημιουργός είναι υποχρεωτικά ιδιωτικός. Αυτή είναι η τυπική περίπτωση στους απαριθμήσιμους τύπους. Κάνοντας τον δημιουργό ιδιωτικό, μπορεί να χρησιμοποιηθεί μόνο από την κλάση για την δημιουργία των απαιτούμενων τιμών, δηλαδή στο παράδειγμά μας, για τη δημιουργία των σταθερών TWO έως ACE. Αν όμως είχαμε ορίσει έναν δημιουργό δημόσιο, τότε οποιοσδήποτε θα μπορούσε να δημιουργήσει τέτοια αντικείμενα με αποτέλεσμα να καταστραφεί η έννοια του απαριθμήσιμου τύπου. Φανταστείτε να δημιουργήσει κανείς τιμή FRIDAY τύπου CardRank. Αν προσπαθήσετε να δηλώσετε δημιουργό σε τύπο enum με δημόσια ή προστατευμένη πρόσβαση, ο μεταγλωττιστής θα διαμαρτυρηθεί. Αν πάλι δεν δηλώσετε προσδιοριστή πρόσβασης, τότε η πρόσβαση δεν θα είναι πρόσβαση πακέτου όπως ισχύει για τις τυπικές κλάσεις αλλά αντίθετα θα καθορισθεί άρρητα ως ιδιωτική.

Έχοντας τον δημιουργό ιδιωτικό έχουμε εξασφαλίσει πως μέσα στο πρόγραμμά μας δεν υπάρχουν άλλα αντικείμενα τύπου CardRank πέραν των δηλωθέντων στον ίδιο τον τύπο. Επομένως, κάθε βαθμός κάρτας στο πρόγραμμά μας είναι μοναδικός, δηλαδή υπάρχει ένα μόνο αντικείμενο TWO, ένα μόνο αντικείμενο THREE και ούτω καθεξής. Σαν συνέπεια, μπορούμε να ελέγχουμε για ισότητα τα αντικείμενα CardRank με χρήση του τελεστή ισότητας. Παρόλο που ο τύπος περιλαμβάνει τη συνάρτηση equals, σε αυτήν την περίπτωση δεν είναι αναγκαία η χρήση της. Αν υποθέσουμε πως έχουμε δύο αναφορές σε ένα αντικείμενο CardRank. Αν και οι δύο αναφορές έχουν ως περιεχόμενο το αντικείμενο TWO ας πούμε, τότε θα αναφέρονται στην ίδια διεύθυνση, μιας και υπάρχει μόνο ένα τέτοιο στιγμιότυπο στην εφαρμογή μας. Επομένως, η σύγκριση με τον τελεστή ισότητας είναι έγκυρη και με δεδομένο πως είναι ταχύτερη από την κλήση της equals την προτιμούμε.

11.8 Εμφωλευμένες κλάσεις

Έχουμε εξηγήσει πως σε ένα αρχείο java μπορεί να περιέχεται μία μόνο δημόσια κλάση και μάλιστα απαιτείται το όνομα του αρχείου να είναι ίδιο με το όνομα της κλάσης (ενότητα 2.2). Τίποτα όμως δεν μας εμποδίζει να τοποθετήσουμε σε ένα αρχείο περισσότερες από μία μη δημόσιες κλάσεις. Για παράδειγμα, ο κώδικας

```
class NonPublic { }

public class NonPublicDemo {

     public static void main(String[] args) {
          NonPublic nP=new NonPublic();
     }
}
```

έστω ότι περιλαμβάνει την κλάση NonPublic στο ίδιο αρχείο με τη δημόσια κλάση NonPublicDemo. Σε αυτό το επίπεδο επιτρέπεται μόνο η πρόσβαση πακέτου για την κλάση NonPublic.

Υπάρχει, επιπλέον η δυνατότητα να ορίσουμε εμφωλευμένες κλάσεις (nested class). Οι εμφωλευμένες κλάσεις είναι δύο κατηγοριών, οι στατικές και οι μη στατικές που ονομάζονται εσωτερικές (inner). Ο κώδικας που ακολουθεί επιδεικνύει τη χρήση εσωτερικής κλάσης.

```
public class Outer {

     private int k = 1;
```

```
        class Inner {

              int getK() {
                    return k;
              }
        }

        public static void main(String[] args) {
              Outer outer = new Outer();
              Outer.Inner iCObject = outer.new Inner();
              System.out.println(outer.k + " " + iCObject.getK());
        }
  }
```

Οι εσωτερικές κλάσεις δέχονται όλους τους ρυθμιστές πρόσβασης, δηλαδή μία εσωτερική κλάση μπορεί να χαρακτηριστεί ως δημόσια, ιδιωτική, προστατευμένη (ενότητα 13.1.2) ή απλώς να διατηρήσει την πρόσβαση πακέτου. Ένα πλεονέκτημα των εσωτερικών κλάσεων είναι ότι έχουν πρόσβαση στα μέλη της κλάσης που ανήκουν ακόμη και όταν αυτά είναι ιδιωτικά. Όπως φαίνεται στον κώδικα, η κλάση Inner προσπελαύνει το ιδιωτικό μέλος k της Outer. Για να δημιουργηθεί αντικείμενο εσωτερικής κλάσης, πρέπει πρώτα να δημιουργηθεί αντικείμενο της περιβάλλουσας μέσα από το οποίο μπορεί να κληθεί ο τελεστής new όπως επιδεικνύει ο παραπάνω κώδικας. Οι εσωτερικές κλάσεις μπορούν να αναπτυχθούν και τοπικά σε επίπεδο συνάρτησης.

Ένας άλλος τύπος εσωτερικής κλάσης είναι η ανώνυμη κλάση (Anonymous class). Θα παρουσιάσουμε την ανώνυμη κλάση στην ενότητα 14.4.5.

Όπως όλα τα μέλη μιας κλάσης, έτσι και τα μέλη-κλάσεις μπορούν να είναι στατικά. Ακολουθεί παράδειγμα χρήσης εμφωλευμένης στατικής κλάσης (nested static class):

```
  public class Outer {

        private int k = 1;

        static class NestedStatic {

              int getK(Outer outer) {
                    return outer.k;
              }
        }

        public static void main(String[] args) {
              Outer.NestedStatic nS = new NestedStatic();
              Outer outer = new Outer();
              System.out.println(outer.k + " " + nS.getK(outer));
        }
  }
```

Η κλάση NestedStatic έχει δηλωθεί ως στατική. Γενικά τα στατικά μέλη δεν έχουν πρόσβαση στα μη στατικά μέλη μιας κλάσης. Ο γενικός αυτός κανόνας ισχύει και για τις κλάσεις. Ένα στατικό μέλος μπορεί να εκτελεστεί αν είναι μέθοδος ή να δημιουργηθεί αν είναι κλάση χωρίς να συσχετίζεται υποχρεωτικά με κάποιο στιγμιότυπο της κλάσης. Επομένως, δεν είναι δυνατόν να προσπελάσει άμεσα τα μη στατικά μέλη. Μπορεί βέβαια η προσπέλαση να επιτευχθεί μέσα από ένα αντικείμενο της κλάσης. Έτσι, η getK εδώ λαμβάνει ως παράμετρο ένα αντικείμενο outer του οποίου εμφανίζει την μεταβλητή k. Όπως οι εσωτερικές κλάσεις, έτσι και οι εμφωλευμένες στατικές μπορούν να συνταχθούν με οποιονδήποτε προσδιοριστή πρόσβασης.

11.9 Λυμένες Ασκήσεις

Παρουσιάζουμε εδώ μια σειρά από λυμένες ασκήσεις που θα βοηθήσουν τον αναγνώστη να εξοικειωθεί με τη λογική της ενσωμάτωσης.

11.9.1 Η κλάση Person

Να υλοποιηθεί η κλάση Person. Κάθε Person διαθέτει όνομα, επώνυμο και φύλο. Για το φύλο χρησιμοποιήστε int έτσι ώστε το 0 να αναπαριστά το γυναικείο φύλλο και το 1 το ανδρικό. Ορίστε τις αναγκαίες συναρτήσεις.

Λύση

```
public class Person {

    private String fName;
    private String sName;
    private int sex;

    public Person(String fName, String sName, int sex) {
        this.fName = fName;
        this.sName = sName;
        this.sex = sex;
    }

    public String getfName() {
        return fName;
    }

    public void setfName(String fName) {
        this.fName = fName;
    }

    public String getsName() {
        return sName;
    }

    public void setsName(String sName) {
        this.sName = sName;
    }

    public int getSex() {
        return sex;
    }

    public void setSex(int sex) {
        this.sex = sex;
    }

    @Override
    public int hashCode() {
        int hash = 5;
        hash = 89 * hash + this.fName.hashCode();
        hash = 89 * hash + this.sName.hashCode();
        hash = 89 * hash + this.sex;
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
```

```
            }
            if (obj == null) {
                return false;
            }
            if (getClass() != obj.getClass()) {
                return false;
            }
            final Person other = (Person) obj;
            if (this.sex != other.sex) {
                return false;
            }
            if (!this.fName.equals(other.fName)) {
                return false;
            }
            return this.sName.equals(other.sName);
        }

        @Override
        public String toString() {
            return "Person{" + "fName=" + fName + ", sName=" + sName + ", sex="
    + (sex==1?"M":"F") + '}';
        }
    }
```

Κώδικας 11.11 *Η κλάση Person*

Στον κώδικα 11.11 παρατίθεται η κλάση Person. Πρόκειται για απλή εφαρμογή της ενσωμάτωσης. Παρόμοιο αποτέλεσμα μπορεί να παραχθεί αυτόματα από το NetBeans ως εξής: Δημιουργήστε μια κλάση Person και προσθέστε τις μεταβλητές στιγμιότυπου. Μεταβείτε στον editor του NetBeans στην κλάση. Με δεξί κλικ ανοίγει ένα μενού. Επιλέξτε Insert Code. Από το υπομενού που θα ανοίξει επιλέξτε σταδιακά την αυτόματη δημιουργία δημιουργών, αναγνωστών, ρυθμιστών, μεθόδου equals, μεθόδου hashcode και μεθόδου toString. Εντοπίστε και μελετήστε τις διαφορές μεταξύ του κώδικα που παράγει το NetBeans και του κώδικα 11.11.

11.9.2 Η κλάση MyInteger

Υλοποιήστε την κλάση MyInteger που περιλαμβάνει τις συναρτήσεις που περιγράφονται παρακάτω και όλες τις αναγκαίες συναρτήσεις.

```
public class MyInteger {
     private int value;
     public MyInteger();
     public MyInteger(int k);
     public MyInteger add(MyInteger i);
     public MyInteger subtract(MyInteger i);
     public MyInteger multiply(MyInteger i);
     public MyInteger divide(MyInteger i);
     public String toString();
     public boolean equals(Object obj);
     public int hashCode();
};
```

Να υπερφορτωθούν οι μέθοδοι setValue, add, subtract, multiply, divide ώστε να δέχονται παραμέτρους τύπου Integer και τύπου int.

Λύση

```
public class MyInteger {

    private int value;
```

```
public MyInteger() {
    value = 0;
}

public MyInteger(int k) {
    value = k;
}

public void setValue(int k) {
    value = k;
}

public void setValue(Integer k) {
    value = k;
}

public void setValue(MyInteger k) {
    value = k.value;
}

public int getValue() {
    return value;
}

public MyInteger add(MyInteger i) {
    return new MyInteger(value + i.value);
}

public MyInteger subtract(MyInteger i) {
    return new MyInteger(value - i.value);
}

public MyInteger multiply(MyInteger i) {
    return new MyInteger(value * i.value);
}

public MyInteger divide(MyInteger i) {
    return new MyInteger(value / i.value);
}

@Override
public String toString() {
    return Integer.toString(value);
}

@Override
public int hashCode() {
    int hash = 7;
    hash = 61 * hash + this.value;
    return hash;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
```

```
        if (getClass() != obj.getClass()) {
            return false;
        }
        final MyInteger other = (MyInteger) obj;
        return this.value == other.value;
    }
}
```

Κώδικας 11.12 *Η κλάση MyInteger*

Η κλάση MyInteger (κώδικας 11.12) εκτός από τις αναγκαίες συναρτήσεις περιλαμβάνει και μια σειρά από συναρτήσεις που υποστηρίζουν τις βασικές πράξεις. Η υλοποίησή τους είναι πολύ απλή. Σε σχέση με την απαίτηση της εκφώνησης να υπερφορτωθούν οι συναρτήσεις setValue, add, subtract, multiply, divide, έχουμε υλοποιήσει την υπερφόρτωση μόνο για την setValue. Παρόμοια μπορείτε να υλοποιήσετε και τις υπόλοιπες αναγκαίες μεθόδους.

11.9.3 Μονάδα Μέτρησης Βάρους

Να υλοποιηθεί η κλάση WeightMeasurementUnit (Μονάδα Μέτρησης Βάρους). Η κλάση διαθέτει δύο μεταβλητές στιγμιότυπου, την description τύπου String που αναπαριστά το λεκτικό της μονάδας και την relationToGr τύπου double που αναπαριστά τη σχέση της μονάδας με το γραμμάριο. Να προστεθεί η συνάρτηση main στην οποία να δημιουργηθούν τρία αντικείμενα, ένα που αναπαριστά το χιλιόγραμμο, ένα για τον τόνο και ένα για το χιλιοστό του γραμμαρίου.

Λύση

```
public class WeightMeasurementUnit {

    private String description;
    private double relationToGr;

    public WeightMeasurementUnit(String lekt, double relationToGr) {
        if (relationToGr <= 0) {
            throw new RuntimeException();
        }
        this.description = lekt;
        this.relationToGr = relationToGr;
    }

    public String getDescription() {
        return description;
    }

    public double getRelationToGr() {
        return relationToGr;
    }

    public void setDescription(String description) {
        this.description = description;
    }


    @Override
    public int hashCode() {
        return Double.hashCode(this.relationToGr);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
```

```
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final WeightMeasurementUnit other = (WeightMeasurementUnit) obj;
        if (Double.compare(this.relationToGr, other.relationToGr) != 0) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "WeightMeasurementUnit{" + "description=" + description + ",
relationToGr=" + relationToGr + '}';
    }

    public static void main(String[] args) {
        WeightMeasurementUnit kgr = new WeightMeasurementUnit("Kgr", 1000),
                metricTon = new WeightMeasurementUnit("Metric Ton", 1000000),
                mgr = new WeightMeasurementUnit("mgr", 1 / 1000d);
        System.out.println(kgr);
        System.out.println(metricTon);
        System.out.println(mgr);
    }
}
```

Κώδικας 11.13 *Κλάση αναπαράστασης της Μονάδας Μέτρησης Βάρους*

Ο κώδικας 11.13 δίνει τον ορισμό της κλάσης WeightMeasurementUnit. Κάθε τέτοια μονάδα προσδιορίζεται από τη σχέση της με το γραμμάριο. Έτσι στον δημιουργό της κλάσης γίνεται έλεγχος ώστε η σχέση της μονάδας με το γραμμάριο να είναι έγκυρη. Στη συνέχεια λαμβάνουν τιμές τα πεδία της κλάσης και δημιουργείται το αντικείμενο. Η ισότητα βασίζεται αποκλειστικά στη μεταβλητή relationToGr, δηλαδή δύο μονάδες μέτρησης βάρους με τη ίδια σχέση με το γραμμάριο θεωρούνται ίσες ανεξάρτητα από τα υπόλοιπα χαρακτηριστικά τους. Αντίστοιχα και η hashCode βασίζεται αποκλειστικά στηνσχέση με το γραμμάριο.

11.9.4 Βάρος

Να υλοποιηθεί η κλάση Weight. Κάθε αντικείμενο της κλάσης διαθέτει δύο μεταβλητές. Η μια εκφράζει το μέτρο του βάρους και είναι double και η άλλη τη μονάδα μέτρησης του βάρους και είναι WeightMeasurementUnit.
Η κλάση διαθέτει τρεις δημιουργούς ως εξής:

```
public Weight(double value, WeightMeasurementUnit mB)
```

Δημιουργεί αντικείμενο με μέτρο value και μονάδα μέτρησης mB.

```
public Weight(Weight w)
```

Δημιουργεί αντίγραφο του w.

```
public Weight(double value, String unitDescription, double
relationToGr)
```

Δημιουργεί αντικείμενο με μέτρο value και μονάδα μέτρησης με περιγραφή unitDescription και σχέση με το γραμμάριο relationToGr.

Πέραν των αναγκαίων συναρτήσεων, η κλάση διαθέτει και τις ακόλουθες συναρτήσεις:

```
public static Weight convert(WeightMeasurementUnit from,
WeightMeasurementUnit to)
```

Μετατρέπει τη μονάδα μέτρησης from μονάδα μέτρησης to. Προσέξτε πως επιστρέφει Weight. Πράγματι, αν για παράδειγμα μετατρέψω το κιλό σε γραμμάρια θα λάβω ως αποτέλεσμα 1000 γραμμάρια, μέγεθος που υποχρεωτικά εκφράζεται σε βάρος. Η συνάρτηση είναι στατική καθώς δεν είναι απαραίτητο να διαθέτουμε αντικείμενο Weight για να μετατρέψουμε μια μονάδα σε μία άλλη.

```
public Weight convert(WeightMeasurementUnit wUnit)
```

Μετατρέπει αυτό το βάρος στην μονάδα wUnit.

```
public Weight add(Weight w)
```

Προσθέτει αυτό το βάρος, δηλαδή το βάρος που την καλεί, με το βάρος w. Επιστρέφει το άθροισμα εκφρασμένο στη μονάδα βάρους που καλεί την add.

Λάβετε υπόψη πως η μέθοδος equals θα πρέπει να επιστρέφει true για ίσα βάρη. Για παράδειγμα, το 1 κιλό είναι ίσο με τα 1000 γραμμάρια. Αντίστοιχα, και η μέθοδος hashCode πρέπει να επιστρέφει την ίδια τιμή για βάρη που είναι ίσα.

Προσθέστε στην κλάση, αυτήν την main

```
public static void main(String[] args) {
     WeightMeasurementUnit          kgr          =          new
     WeightMeasurementUnit("kgr", 1000);
     WeightMeasurementUnit          mgr          =          new
     WeightMeasurementUnit("mgr", 1 / 1000d);
     Weight w1 = new Weight(1, kgr);
     Weight w2 = new Weight(100, mgr);
     Weight w3 = w1.add(w2);
     Weight w4 = w2.add(w1);
     System.out.println(w3);
     System.out.println(w4);
     System.out.println(w3.equals(w4));
     System.out.println(w3.hashCode() == w4.hashCode());
}
```

Ελέγξτε ώστε η έξοδος να είναι σύμφωνη με

```
1.0001 kgr
1000100.0 mgr
true
true
```

Λύση

```
public class Weight {

    private double value;
    private WeightMeasurementUnit weightUnit;

    public Weight(double value, String unitDescription,
                  double relationToGr) {
        this(value, new WeightMeasurementUnit(unitDescription,
                  relationToGr));
    }

    public Weight(double value, WeightMeasurementUnit mB) {
        this.value = value;
```

```
        this.weightUnit = mB;
    }

    public Weight(Weight w) {
        this(w.value, w.weightUnit);
    }

    public static Weight convert(WeightMeasurementUnit from,
            WeightMeasurementUnit to) {
        double valueTo = from.getRelationToGr() / to.getRelationToGr();
        return new Weight(valueTo, to);
    }

    public Weight convert(WeightMeasurementUnit wUnit) {
        Weight rVal = convert(weightUnit, wUnit);
        rVal.value *= value;
        return rVal;
    }

    public Weight add(Weight w) {
        //Weight w1 = new Weight(w);
        if (!weightUnit.equals(w.weightUnit)) {
            w = w.convert(weightUnit);
        }
        return new Weight(value + w.value, weightUnit);
    }

    public void setValue(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    public WeightMeasurementUnit getWeightUnit() {
        return weightUnit;
    }

    public void setWeightUnit(WeightMeasurementUnit weightUnit) {
        this.weightUnit = weightUnit;
    }

    @Override
    public int hashCode() {
        Weight w = convert(new WeightMeasurementUnit("gr", 1));
        int hash = 7;
        hash = 23 * hash + Double.hashCode(w.value);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
```

```
        }
        Weight other = (Weight) obj;
        other = other.convert(this.weightUnit);
        return Double.compare(this.value, other.value) == 0;
    }

    @Override
    public String toString() {
        return value + " " + weightUnit.getDescription();
    }

    public static void main(String[] args) {
        WeightMeasurementUnit kgr = new WeightMeasurementUnit("kgr", 1000);
        WeightMeasurementUnit mgr = new WeightMeasurementUnit("mgr", 1 / 1000d);
        Weight w1 = new Weight(1, kgr);
        Weight w2 = new Weight(100, mgr);
        Weight w3 = w1.add(w2);
        Weight w4 = w2.add(w1);
        System.out.println(w3);
        System.out.println(w4);
        System.out.println(w3.equals(w4));
        System.out.println(w3.hashCode() == w4.hashCode());
    }
}
```

Κώδικας 11.14 *Η κλάση Weight*

Για να μετατρέψω μια μονάδα βάρους σε μια άλλη θα πρέπει να αξιοποιήσω την σχέση κάθε μονάδας με το γραμμάριο. Έστω ότι θέλω να μετατρέψω τη μονάδα βάρους from στην to. Η from σε γραμμάρια είναι ένα βάρος με τιμή from.relationToGr και μονάδα το γραμμάριο. Αντίστοιχα, η to σε γραμμάρια είναι ένα βάρος με μέτρο to.relationToGr και μονάδα το γραμμάριο. Τώρα και οι δύο μονάδες είναι εκφρασμένες σε βάρη με μονάδα το γραμμάριο. Επομένως, η διαίρεση from.relationToGr/to.relationToGr μας δίνει το μέτρο του βάρους που εκφράζει τη μονάδα from στην μονάδα to. Αυτή είναι η λογική της convert που μετατρέπει από μια μονάδα σε άλλη, δηλαδή της convert(WeightMeasurementUnit, WeightMeasurementUnit) του κώδικα 11.14.

Για να μετατρέψω τη μονάδα μέτρησης ενός βάρους, στη μονάδα μέτρησης, έστω wUnit, μετατρέπω πρώτα τη μονάδα μέτρησης του βάρους, δηλαδή την weightUnit στην wUnit. Στη συνέχεια, το μέτρο του βάρους που λαμβάνω ως αποτέλεσμα της μετατροπής των μονάδων, το πολλαπλασιάζω με το μέτρο αυτού του βάρους. Σε αυτήν τη λογική βασίζεται η convert(WeightMeasurementUnit) του κώδικα 11.14.

Στη συνάρτηση add ελέγχω αν τα βάρη που πρόκειται να προστεθούν εκφράζονται στην ίδια μονάδα βάρους. Αν όχι μετατρέπω το βάρος που είναι παράμετρος της add στη μονάδα του βάρους που καλεί την add. Εφόσον τα δύο βάρη εκφράζονται στην ίδια μονάδα, η πρόσθεσή τους γίνεται απλώς με την πρόσθεση των μέτρων τους. Το νέο βάρος έχει ως μονάδα την μονάδα του βάρους που κάλεσε την add. Έτσι, η κλήση w1.add(w2) δεν θα επιστρέψει ακριβώς το ίδιο αποτέλεσμα με την κλήση w2.add(w1), καθώς στην πρώτη περίπτωση το βάρος εκφράζεται στη μονάδα της w1 και στη δεύτερη στη μονάδα της w2. Ωστόσο, τα δύο αυτά αθροίσματα παρότι εκφράζονται σε διαφορετικές μονάδες βάρους, αντιπροσωπεύουν ίσα βάρη, πράγμα που πρέπει να επιβεβαιώνεται και από τη μέθοδο equals της κλάσης.

Η μέθοδος equals θα πρέπει να επιστρέφει true για βάρη που είναι ίσα μεταξύ τους, π.χ. το 1 kgr με τα 1000 gr, πρέπει να θεωρούνται ίσα. Έτσι μετατρέπουμε την παράμετρό της στη μονάδα βάρους του αντικειμένου που την καλεί και στη συνέχεια συγκρίνουμε τα μέτρα των δύο βαρών.

Παρόμοια, η hashCode πρέπει να επιστρέφει τον ίδιο κωδικό για δύο βάρη εφόσον η equals τα θεωρεί ίσα. Έτσι μετατρέπουμε κάθε βάρος σε γραμμάρια και παράγουμε κωδικό βασισμένοι στον αριθμό των γραμμαρίων ενός βάρους.

11.9.5 Η κλάση Complex

Κάθε μιγαδικός αριθμός αποτελείται από ένα ζεύγος πραγματικών και συμβολίζεται ως (a,b) ή a+bi, όπου ο πραγματικός a αποτελεί το πραγματικό μέρος του μιγαδικού και ο b το φανταστικό μέρος.

Η πρόσθεση μεταξύ δύο μιγαδικών δίνεται από τον τύπο

$$(x + yi) + (u + vi) = (x + u) + (y + v)i$$

Η αφαίρεση δίνεται από τον τύπο

$$(x + yi) - (u + vi) = (x - u) + (y - v)i$$

Ο πολλαπλασιασμός από τον τύπο

$$(x + yi)(u + vi) = (xu - yv) + (\mathrm{x}v - yu)i$$

Ο αντίστροφος ενός μιγαδικού δίνεται από τον τύπο

$$\frac{1}{x + yi} = \frac{x}{x^2 + y^2} - \frac{y}{x^2 + y^2}i$$

Να υλοποιηθεί η κλάση Complex των μιγαδικών αριθμών. Να προστεθούν μέθοδοι για την πρόσθεση, την αφαίρεση, τον πολλαπλασιασμό και τη διαίρεση μιγαδικών. Να υποστηρίζονται και οι δύο συμβολισμοί των μιγαδικών. Να προστεθεί συνάρτηση main στην οποία να διαιρείται ο 2+3i με τον 4+0i και ο 20-4i με τον 3+2i.

Σημείωση: Τα αποτελέσματα των ζητούμενων διαιρέσεων είναι 0.5+0.75i και 4.0-4.0i, αντίστοιχα.

Λύση

```
public class Complex {

    public static int format = 1;
    private double r;
    private double f;

    public Complex(double r, double f) {
        this.r = r;
        this.f = f;
    }

    public Complex(Complex c) {
        this(c.r, c.f);
    }

    public double getR() {
        return r;
    }

    public void setR(double r) {
        this.r = r;
    }

    public double getF() {
        return f;
    }

    public void setF(double f) {
        this.f = f;
    }

    @Override
```

```
public int hashCode() {
    return Double.hashCode(r) + Double.hashCode(f);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Complex other = (Complex) obj;
    if (Double.compare(this.r, other.r) != 0) {
        return false;
    }
    return Double.compare(this.f, other.f) == 0;
}

@Override
public String toString() {
    if (r==0 && f==0) return "0";
    String rVal="";
    if (format == 1) {
        if (r!=0)
            rVal+=r;  //8i, 2-81
        if (f>0)
            rVal+=(r!=0?"+":"")+f+"i";
        else
            if (f<0)
                rVal+=f+"i";
    }
    else {
        rVal="("+r+","+f+")";
    }
    return rVal;
}

public Complex add(Complex c) {
    double real=r+c.r;
    double imagin=f+c.f;
    return new Complex(real, imagin);
}

public Complex subtract(Complex c) {
    double real=r-c.r;
    double imagin=f-c.f;
    return new Complex(real, imagin);
}

public Complex multiply(Complex c) {
    double real=r*c.r-f*c.f;
    double imagin=r*c.f+f*c.r;
    return new Complex(real, imagin);
}

public Complex divide(Complex c) {
    if (c.r==0 && c.f==0)
```

```
                throw new RuntimeException();
            double real=c.r/(Math.pow(c.r, 2)+Math.pow(c.f, 2));
            double imagin=c.f/(Math.pow(c.r, 2)+Math.pow(c.f, 2));
            Complex divisor= new Complex(real,-imagin);
            return multiply(divisor);
        }

        public static void main(String[] args) {
            Complex.format=1;
            Complex c1=new Complex(2,3);
            Complex c2=new Complex(4,0);
            System.out.println(c1.divide(c2));
            c1=new Complex(20,-4);
            c2=new Complex(3,2);
            System.out.println(c1.divide(c2));
            System.out.println(Double.doubleToLongBits(c1.r) ^
    Double.doubleToLongBits(c1.r)>>> 32);
        }
    }
```

Κώδικας 11.15 *Η κλάση Complex των Μιγαδικών*

Η κλάση Complex στον κώδικα 11.15 διαθέτει τη στατική μεταβλητή format και δύο μεταβλητές στιγμιότυπου, τις r και f. Η format αξιοποιείται από την toString η οποία ανάλογα με την τιμή της format ρυθμίζει ποιος από τους δύο συμβολισμούς των αντικειμένων της Complex θα χρησιμοποιείται, ενώ τα μέλη r και f αναπαριστούν το πραγματικό και φανταστικό μέρος του μιγαδικού. Στη συνέχεια περιλαμβάνει δύο δημιουργούς. Ο πρώτος είναι ο τυπικός δημιουργός που κατασκευάζει μιγαδικούς με είσοδο δύο πραγματικών, ενώ ο δεύτερος είναι δημιουργός αντιγραφής. Οι αναγνώστες, οι ρυθμιστές, η equals και η hashCode της κλάσης είναι τυπικές οπότε δεν τις σχολιάζουμε παραπέρα. Η μέθοδος toString είναι λίγο πιο περίπλοκη, τόσο για να υποστηρίζει τους διαφορετικούς συμβολισμούς των αντικειμένων όσο και για να υποστηρίζει ειδικές περιπτώσεις μιγαδικών όπως οι καθαροί φανταστικοί, π.χ., (0, 5), οι καθαροί πραγματικοί, π.χ. (2, 0) και οι μιγαδικοί με αρνητικό φανταστικό μέρος, π.χ. (2, -8). Οι μέθοδοι add, subtract και multiply αποτελούν καθαρή εφαρμογή των αντίστοιχων τύπων που δόθηκαν στην εκφώνηση. Η μέθοδος divide ελέγχει αρχικά αν ζητείται διαίρεση με το 0. Θεωρούμε εδώ πως η διαίρεση με το 0 δεν ορίζεται. Επομένως, αν μια εφαρμογή επιχειρήσει διαίρεση με το 0, ο κώδικας θα αποκριθεί με την παραγωγή εξαίρεσης, Στη συνέχεια η διαίρεση μεταξύ δύο μιγαδικών x/y γίνεται αφού υπολογίσουμε τον αντίστροφο του διαιρέτη, z=1/y και πολλαπλασιάσουμε το x με το z.

11.9.6 Παιχνίδι ζαριών

Τρεις παίκτες, ο John, ο Jim και ο Jack συναντήθηκαν προκειμένου να παίξουν ένα παιχνίδι ζαριών αποτελούμενο από παρτίδες με τους ακόλουθους κανόνες: Σε κάθε παρτίδα, οι παίκτες ρίχνουν εναλλάξ από 10 ζαριές. Κάθε φορά που ένας παίκτης φέρνει άρτιο αποτέλεσμα αυξάνεται κατά 1 το σκορ του. Στην περίπτωση που δύο ή περισσότεροι παίκτες ισοβαθμήσουν, η παρτίδα επαναλαμβάνεται. Σε κάθε επανάληψη, τα σκορ των παικτών ενημερώνονται αθροιστικά. Όταν στο τέλος μιας παρτίδας, όλοι οι παίκτες έχουν διαφορετικό σκορ, το παιχνίδι λήγει και νικητής ανακηρύσσεται ο παίκτης με το μεγαλύτερο σκορ.

Οι παίκτες όμως δεν είναι το ίδιο τυχεροί. Ο John φέρνει κάθε δυνατό αποτέλεσμα με πιθανότητα 1/6, ο Jim δεν φέρνει ποτέ 5 και στη θέση του φέρνει 6 με πιθανότητα 1/3, ο Jack δεν φέρνει ποτέ 6 και στη θέση του φέρνει 5 με πιθανότητα 1/3. Όλα τα άλλα αποτελέσματα, τόσο ο Jim όσο και ο Jack τα φέρνουν με πιθανότητα 1/6.

Να υλοποιηθεί η κλάση DiceGame που διαθέτει τη μέθοδο public DicePlayer play() που προσομοιώνει ένα παιχνίδι ζαριών και επιστρέφει τον νικητή. Να προστεθεί main που καλεί την play() και εμφανίζει τον νικητή και το σκορ του.

Υπόδειξη: Δημιουργήστε μια κλάση που αναπαριστά έναν παίκτη ζαριού. Δημιουργήστε τρία αντικείμενα της κλάσης, από ένα για τον John, τον Jim και τον Jack.

Λύση

Για την υλοποίηση αυτής της άσκησης, χρησιμοποιούμε συνολικά τρεις κλάσεις, την Lib, την DicePlayer και την DiceGame. Η Lib περιλαμβάνει στατικές συναρτήσεις και σταθερές γενικού χαρακτήρα, η DicePlayer μοντελοποιεί τον παίκτη ζαριού και η DiceGame το παιχνίδι ζαριών έτσι όπως περιγράφηκε στην εκφώνηση.

```
import java.util.Random;

public class Lib {
    public final static double EPSILON=0.000001;
    public final static Random gen=new Random();

    public static boolean approximateEquals(double f1, double f2) {
        return Math.abs(f2 - f1) < EPSILON;
    }

    public static double sum(double[] t) {
        double sum=0;
        for (double d:t)
            sum+=d;
        return sum;
    }
}
```

Κώδικας 11.16 *Η κλάση Lib*

Όπως φαίνεται στον κώδικα 11.16, στην Lib ορίζονται στατικές μεταβλητές και συναρτήσεις γενικού σκοπού. Πιο συγκεκριμένα, ορίζεται μια συνάρτηση προσεγγιστικής ισότητας, μια συνάρτηση που υπολογίζει το άθροισμα των στοιχείων μονοδιάστατου πίνακα, μια γεννήτρια τυχαίων αριθμών και η σταθερά EPSILON που αξιοποιείται από την προσεγγιστική ισότητα. Όπως θα δούμε, όλα τα μέλη της Lib χρησιμοποιούνται από την κλάση DicePlayer. Με αυτήν την έννοια θα μπορούσαν να είναι μέλη της DicePlayer. Τοποθετήθηκαν όμως σε χωριστή κλάση λόγω του γενικού χαρακτήρα τους. Η προσεγγιστική ισότητα για παράδειγμα μπορεί να αξιοποιηθεί και από άλλα πρότζεκτ που δεν έχουν καμία σχέση με το παιχνίδι ζαριών. Ο σχεδιασμός που προτείνουμε μας διευκολύνει ώστε σε αυτά, τα προτζεκτ να εισάγουμε μόνο την Lib και όχι την DicePlayer.

```
import java.util.Arrays;
import java.util.Objects;

public class DicePlayer {

    private final double[] distribution;
    private final String name;

    public DicePlayer(String name, double[] distribution) {

        if (distribution.length != 6
                || !Lib.approximateEquals(Lib.sum(distribution), 1)) {
            throw new RuntimeException();
        }
        this.name = name;
        this.distribution = distribution;
    }

    public DicePlayer(String name) {
        this(name, new double[]{1d / 6, 1d / 6, 1d / 6, 1d / 6, 1d / 6, 1d
/ 6,});
    }

    public int roll() {
        double value = Lib.gen.nextDouble();
        double sum = 0;
```

```
            for (int i = 0; i < distribution.length; i++) {
                sum += distribution[i];
                if (value <= sum) {
                    return i + 1;
                }
            }
            return 0;
        }

        public String getName() {
            return name;
        }

        @Override
        public int hashCode() {
            return Arrays.hashCode(this.distribution)+this.name.hashCode();
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) {
                return true;
            }
            if (obj == null) {
                return false;
            }
            if (getClass() != obj.getClass()) {
                return false;
            }
            final DicePlayer other = (DicePlayer) obj;
            if (!Objects.equals(this.name, other.name)) {
                return false;
            }
            return Arrays.equals(this.distribution, other.distribution);
        }
    }
```

Κώδικας 11.17 *Η κλάση DicePlayer*

Ο δημιουργός της κλάσης DicePlayer του κώδικα 11.17 παρέχει τη δυνατότητα να κατασκευάζονται παίκτες με διαφορετικά χαρακτηριστικά ζαριάς. Λαμβάνει 2 παραμέτρους, μια για το όνομα του παίκτη και μια για την κατανομή που καθορίζει τα αποτελέσματα των ρίψεων ζαριού για τον συγκεκριμένο παίκτη. Ο δημιουργός ελέγχει ώστε το σύνολο των πιθανοτήτων για να φέρει ένα αποτέλεσμα από 1 έως 6 ο παίκτης να είναι ίσο με 1. Επίσης, παρέχεται ένας δημιουργός με παράμετρο μόνο το όνομα του παίκτη που κατασκευάζει έναν παίκτη με ίση πιθανότητα για κάθε πιθανή ζαριά.

Προσέξτε πως τα μέλη name και distribution είναι final ενώ δεν παρέχονται για αυτά συναρτήσεις ρυθμιστές. Έτσι μετά τη δημιουργία ενός παίκτη δεν υπάρχει η δυνατότητα να αλλάξει το όνομά του ή τα χαρακτηριστικά της ζαριάς του.

Η συνάρτηση roll βασίζεται στον αλγόριθμο που περιγράφεται στον κώδικα 8.4.

Οι equals και hashCode είναι σχετικά απλές, κρίσιμες όμως στην συγκεκριμένη εφαρμογή, καθώς όπως θα δούμε στην DiceGame χρησιμοποιείται ένα HashMap που τις αξιοποιεί.

```
import java.util.HashMap;

public class DiceGame {

    private final int noOfPlayers;
    private final int noOfRoll;

    private final DicePlayer john = new DicePlayer("John");
```

```
    private final DicePlayer jim = new DicePlayer("Jim", new double[]{1d / 6,
1d / 6, 1d / 6, 1d / 6, 0, 2d / 6});
    private final DicePlayer jack = new DicePlayer("Jack", new double[]{1d /
6, 1d / 6, 1d / 6, 1d / 6, 2d / 6, 0});

    private final HashMap<DicePlayer, Integer> scores = new HashMap<>();

    public DiceGame(int noOfPlayers, int noOfRoll) {
        this.noOfPlayers = noOfPlayers;
        this.noOfRoll = noOfRoll;
    }

    private void updScores(DicePlayer player) {
        Integer score = scores.get(player);
        if (score == null) {
            score = 0;
        }
        scores.put(player, score + 1);
    }

    private void aLot() {

        for (int i = 0; i < noOfRoll; i++) {
            int rslt = john.roll();
            if (rslt % 2 == 0) {
                updScores(john);
            }

            rslt = jim.roll();
            if (rslt % 2 == 0) {
                updScores(jim);
            }

            rslt = jack.roll();
            if (rslt % 2 == 0) {
                updScores(jack);
            }
        }
    }

    private boolean withdraw() {
        int score1 = scores.get(john),
                score2 = scores.get(jim),
                score3 = scores.get(jack);
        return (score1 == score2 || score2 == score3 || score1 == score3);
    }

    private DicePlayer max() {
        int scoreJohn = scores.get(john) != null ? scores.get(john) : 0;
        int scoreJim = scores.get(jim) != null ? scores.get(jim) : 0;
        int scoreJack = scores.get(jack) != null ? scores.get(jack) : 0;
        int max = scoreJohn;
        DicePlayer rVal = john;
        if (scoreJim > max) {
            max = scoreJim;
            rVal = jim;
        }
        if (scoreJack > max) {
            rVal = jack;
        }
        return rVal;
```

```
    }

    public DicePlayer play() {
        do {
            aLot();
        } while (withdraw());
        return max();
    }

    public int getNoOfPlayers() {
        return noOfPlayers;
    }

    public int getNoOfRoll() {
        return noOfRoll;
    }

    public static void main(String[] args) {
        DiceGame diceGame = new DiceGame(3, 10);
        DicePlayer winner = diceGame.play();
        System.out.println("Winner is " + winner.getName() + " Total Score "
+ diceGame.scores.get(winner));
    }
}
```

Κώδικας 11.18 *Η κλάση DiceGame*

Η μέθοδος play στον κώδικα 11.18 επαναλαμβάνει μια παρτίδα ζαριών ενόσω εντοπίζεται ισοπαλία. Με τη λήξη της ισοπαλίας επιστρέφει τον παίκτη που πέτυχε το μεγαλύτερο σκορ. Στα ιδιωτικά μέλη της κλάσης περιλαμβάνεται το μέλος scores τύπου HashMap<DicePlayer, Integer>. Πρόκειται για μια κλάση που ανήκει στο πακέτο java.util και πιο συγκεκριμένα για μία γενίκευση. Αναλυτικά θα μιλήσουμε για τις γενικεύσεις στην ενότητα 17. Για την ώρα μας αρκεί να γνωρίζουμε πως χρησιμοποιούμε το scores στα πλαίσια της άσκησής μας. Το scores αποτελεί ένα ευρετήριο του οποίου το κλειδί είναι τύπου DicePlayer και η τιμή τύπου Integer. Με άλλα λόγια, συσχετίζει έναν παίκτη με μια ακέραια τιμή που αντιπροσωπεύει το σκορ του παίκτη. Παράλληλα, έχουν δηλωθεί και αρχικοποιηθεί και οι τρεις παίκτες της άσκησης.

Η συνάρτηση aLot προσομοιώνει μια παρτίδα. Μέσα σε έναν βρόγχο 10 επαναλήψεων, κάθε παίκτης ρίχνει μια ζαριά. Αν το αποτέλεσμα είναι άρτιο καλεί την updScores και ενημερώνει το σκορ του παίκτη. Η updateScores με την σειρά της ελέγχει αν στο scores υπάρχει το score του παίκτη. Αν δεν βρεθεί η get επιστρέφει null. Σε αυτήν την περίπτωση αρχικοποιείται το σκορ του παίκτη και ενημερώνεται η scores. Σημειώστε πως η put αν δεν βρεθεί εγγραφή με αυτόν τον παίκτη εισάγει νέα εγγραφή, ενώ αν βρεθεί, ενημερώνει την υπάρχουσα.

Η withdraw επιστρέφει true αν εντοπίσει έστω και ένα ζεύγος ισοβαθμούντων παικτών. Τέλος η max επιστρέφει τον παίκτη με το μεγαλύτερο σκορ από τους τρεις.

Η άσκηση μπορεί να λυθεί και χωρίς την HashMap, ωστόσο είναι χρήσιμο να ανακαλύπτουμε νέες κλάσεις από τη βιβλιοθήκη της Java.

11.9.7 Η Τράπουλα

Μια τράπουλα αποτελείται από 52 κάρτες, δηλαδή από όλες τις τυπικές κάρτες χωρίς μπαλαντέρ. Κάθε κάρτα έχει δύο χαρακτηριστικά, το χρώμα και τον βαθμό. Το χρώμα λαμβάνει τις τιμές καρό, κούπα, σπαθί και μπαστούνι. Ο δε βαθμός λαμβάνει τις τιμές 2,3,…, Βαλές, Ντάμα, Ρήγας, Άσος.

Να υλοποιηθεί η κλάση Deck (Τράπουλα). Το χρώμα και ο βαθμός κάθε κάρτας να κωδικοποιηθούν ως απαριθμήσιμοι τύποι.

Να υλοποιηθεί εφαρμογή που δημιουργεί δύο τράπουλες. Ανοίγει την πρώτη, δηλαδή εμφανίζει όλες τις κάρτες της, τέσσερις ανά γραμμή με την ακόλουθη σειρά: Οι κάρτες ενός βαθμού εμφανίζονται στην ίδια γραμμή πρώτα οι κούπες, μετά τα μπαστούνια ακολουθούμενα από τα σπαθιά και τέλος τα καρό, όπως δείχνει το παρακάτω απόσπασμα εξόδου.

```
TWO HEARTS TWO SPADES TWO CLUBS TWO DIAMONDS
THREE HEARTS THREE SPADES THREE CLUBS THREE DIAMONDS
…
…
TEN HEARTS TEN SPADES TEN CLUBS TEN DIAMONDS
JACK HEARTS JACK SPADES JACK CLUBS JACK DIAMONDS
QUEEN HEARTS QUEEN SPADES QUEEN CLUBS QUEEN DIAMONDS
KING HEARTS KING SPADES KING CLUBS KING DIAMONDS
ACE HEARTS ACE SPADES ACE CLUBS ACE DIAMONDS
```

Στη συνέχεια, ανακατέψτε τη δεύτερη τράπουλα και ανοίξτε τις κάρτες της.

Υπόδειξη: Προγραμματίστε κατάλληλα τη μέθοδο toString ώστε να εμφανίζει τις κάρτες της τράπουλας, τέσσερις ανά γραμμή με τη σειρά που είναι τοποθετημένες οι κάρτες στην τράπουλα.

Λύση
Έναν απαριθμήσιμο τύπο για να αναπαραστήσουμε τον βαθμό μιας κάρτας τράπουλας έχουμε δώσει στην ενότητα 11.6. Πρόκειται για τον τύπο CardRank. Ο τύπος είναι κατάλληλος για τις απαιτήσεις της άσκησης, οπότε θα χρησιμοποιήσουμε αυτόν. Ένας κατάλληλος τύπος για την αναπαράσταση του χρώματος δίνεται στον κώδικα 11.19:

```
public enum Suit { //Κώδικας 11.19
  HEARTS,
  SPADES,
  CLUBS,
  DIAMONDS;
}
```

Κώδικας 11.19 *Αναπαράσταση του χρώματος κάρτας τράπουλας*

Στη συνέχεια, η κλάση που αναπαριστά κάρτες μπορεί να αναπτυχθεί όπως δείχνει ο κώδικας 11.20:

```
public class Card {

    private final Suit suit;
    private final CardRank cardRank;

    public Card(CardRank cardRank, Suit suit) {
        this.cardRank = cardRank;
        this.suit = suit;
    }

    public Suit getSuit() {
        return suit;
    }

    public CardRank getCardRank() {
        return cardRank;
    }

    @Override
    public String toString() {
        return this.cardRank + " " + this.suit;
    }
}
```

Κώδικας 11.20 *Κλάση αναπαράστασης κάρτας τράπουλας*

Ας σημειωθεί πως σε αυτήν τη λύση, για οικονομία χώρου, συμπεριλαμβάνουμε μόνο συναρτήσεις που είναι αναγκαίες για τη συγκεκριμένη εφαρμογή. Μπορείτε να συμπληρώσετε τις υπόλοιπες αναγκαίες συναρτήσεις στις κλάσεις μόνοι σας για εξάσκηση.

Όσον αφορά την κλάση Card, είναι μια πολύ απλή κλάση που συνδυάζει ένα αντικείμενο Suite και ένα CardRank. Διαθέτει αναγνώστες αλλά όχι ρυθμιστές, καθώς τόσο η cardRank όσο και η suite είναι final. Αυτή η επιλογή υπαγορεύεται από την ανάγκη να μην αλλάζει ο βαθμός ή το χρώμα μιας κάρτας μετά τη δημιουργία της, κάτι που ισχύει και στην πραγματικότητα του φυσικού κόσμου.

Τώρα μπορούμε να δούμε και την κλάση που αναπαριστά τράπουλες:

```
import java.util.Random;

public class Deck {

    private final Card[] deck;
    private int cardsInDeck;

    public Deck() {
        deck = new Card[52];
        cardsInDeck = 0;
        for (CardRank r : CardRank.values()) {
            for (Suit s : Suit.values()) {
                deck[cardsInDeck++] = new Card(r, s);
            }
        }
    }

    public void shuffle() {
        Random r = new Random();

        for (int i = 0; i < deck.length; i++) {
            int rIdx = r.nextInt(deck.length);
            Card tmp = deck[i];
            deck[i] = deck[rIdx];
            deck[rIdx] = tmp;
        }
    }

    public boolean empty() {
        return cardsInDeck == 0;
    }

    public Card pickCard() {
        cardsInDeck--;
        return deck[52 - cardsInDeck - 1];
    }

    @Override
    public String toString() {
        if (cardsInDeck == 0) {
            return "Empty Deck";
        }
        String rVal = "";
        for (int i = 52 - cardsInDeck; i < 52; i++) {
            if (i > 52 - cardsInDeck && i % 4 == 0) {
                rVal += "\n";
            }
            rVal += deck[i] + " ";
        }
        return rVal;
    }
```

```
    public static void main(String[] args) {
        Deck d1 = new Deck();
        Deck d2 = new Deck();

        for (int i = 0; i < 24; i++) {
            d1.pickCard();
        }
        System.out.println(d1.cardsInDeck);
        System.out.println(d1);
        System.out.println("");

        d2.shuffle();
        System.out.println(d2);
    }
}
```

Κώδικας 11.21 *Η κλάση Deck*

Όπως φαίνεται στον κώδικα 11.21, η κλάση Deck αποθηκεύει τις κάρτες στον πίνακα-ιδιωτικό μέλος deck. Επίσης, διαθέτει το ιδιωτικό μέλος cardsInDeck το οποίο παρακολουθεί πόσες κάρτες εναπομένουν στην τράπουλα. Τόσο η μεταβλητή deck όσο και η cardsInDeck λαμβάνουν τιμή στον δημιουργό της κλάσης. Εκεί δημιουργούνται οι κάρτες που εισάγονται στην τράπουλα με τη σειρά που μας ζητάει η εκφώνηση να εμφανιστούν οι κάρτες μιας μη ανακατεμένης τράπουλας. Με την ολοκλήρωση της εκτέλεσης του δημιουργού, η τράπουλα διαθέτει και τα 52 φύλλα στην απαιτούμενη σειρά και η μεταβλητή cardsInDeck έχει την τιμή 52. Η συνάρτηση isEmpty απλά επιστρέφει true όταν η μεταβλητή cardsInDeck είναι ίση με το 0. Η cardsInDeck μειώνεται από τη μέθοδο pickCard η οποία επιστρέφει μια κάρτα από την τράπουλα. Ποια κάρτα όμως ακριβώς επιστρέφει δεδομένου ότι όλα τα αντικείμενα τύπου Card παραμένουν ως στοιχεία του πίνακα deck;

Όπως φαίνεται στον κώδικα 11.21, η pickCard σηκώνει την κάρτα στη θέση 52 - cardsInDeck – 1 του deck. Πράγματι, αν υποθέσουμε πως η cardsInDeck έχει την τιμή 28, δηλαδή έχουν ληφθεί 24 κάρτες (μέσω της pickCard) από την τράπουλα των 52 φύλλων, σε ποια θέση βρίσκεται η επόμενη διαθέσιμη κάρτα; Εφόσον, έχουν ληφθεί 24 κάρτες από την τράπουλα, η επόμενη διαθέσιμη κάρτα βρίσκεται στη θέση 25, δηλαδή στη θέση 52 μείον 28 που είναι η τιμή της cardsInDeck μείον 1 λόγω του ότι ο πίνακας είναι zero based.

Η μέθοδος toString ελέγχει πρώτα αν η τράπουλα είναι άδεια οπότε επιστρέφει τη σειρά “Empty Deck”. Αν δεν επιστρέψει “Empty Deck”, συνεχίζει και στήνει στην rVal τις κάρτες της τράπουλας. Πρέπει ωστόσο η rVal να συμπεριλαμβάνει μόνο τις κάρτες που έχουν απομείνει. Αν έχουν για παράδειγμα απομείνει 5 κάρτες, δηλαδή cardsInDeck==5, δεδομένου ότι η pickCard επιστρέφει την πρώτη εναπομένουσα κάρτα, αυτές είναι οι 5 κάρτες που είναι τοποθετημένες τελευταίες στον πίνακα deck. Επομένως, η πρόσθεση των καρτών στην rVal θα πρέπει να αρχίσει από τη θέση 52-cardsInDeck του deck.

Δεδομένου ότι η εκφώνηση μας ζητάει να εμφανίζονται τέσσερις κάρτες από έναν βαθμό (εννοείται εφόσον υπάρχουν και δεν έχουν ληφθεί ήδη από την pickCard) σε μια γραμμή, ελέγχουμε και εφόσον η μεταβλητή i διαιρείται ακριβώς με το τέσσερα και είναι μεγαλύτερη από την τιμή 52-cardsInDeck, δηλαδή δεν πρόκειται για την πρώτη γραμμή, τότε προσθέτουμε τον χαρακτήρα new line στην rVal. Στο τέλος της επαναληπτικής διαδικασίας, η rVal έχει το απαιτούμενο περιεχόμενο οπότε την επιστρέφουμε.

Τέλος, έχοντας στη διάθεσή μας την κλάση Deck και τις λοιπές βοηθητικές κλάσεις, η εφαρμογή που ζητά η εκφώνηση είναι απλή και παρουσιάζεται στον κώδικα 11.22.

```
public class App {

    public static void main(String[] args) {
        Deck d1 = new Deck();
        Deck d2 = new Deck();
        System.out.println(d1);
        d2.shuffle();
        System.out.println("");
        System.out.println(d2);
```

```
        }
    }
```

Κώδικας 11.22 *Η εφαρμογή της άσκησης 11.9.7*

11.10 Ασκήσεις προς λύση

11.10.1 Η κλάση των Πρώτων Αριθμών

Να υλοποιηθεί η κλάση PrimeNumber. Κάθε αντικείμενο της κλάσης αναπαριστά έναν πρώτο αριθμό. Η κλάση διαθέτει τα ακόλουθα μέλη:

```
PrimeNumber(int val)
```

Ο δημιουργός κάνει έλεγχο ώστε το αντικείμενο να αντιπροσωπεύει πράγματι έναν πρώτο αριθμό.

```
public static boolean isPrime(int val)
```

Ελέγχει αν η ακέραιη παράμετρός της είναι πρώτος αριθμός.

```
public static int getNext(int val)
```

Επιστρέφει τον μικρότερο αριθμό που είναι πρώτος και μεγαλύτερος από την ακέραια παράμετρό της.

```
PrimeNumber getNext()
```

Επιστρέφει τον μικρότερο αριθμό που είναι πρώτος και μεγαλύτερος από το αντικείμενο που την καλεί.
Να προστεθούν όλες οι αναγκαίες μέθοδοι. Να προστεθεί main που εμφανίζει τους 100 πρώτους αριθμούς.

11.10.2 Η κλάση SetOfStrings

Η κλάση SetOfStrings αναπαριστά σύνολα από Strings. Σε ένα σύνολο δεν μπορεί να υπάρχει 2 φορές το ίδιο στοιχείο. Η κλάση περιλαμβάνει τα ακόλουθα μέλη:

```
public SetOfStrings(int size)
```

Δημιουργεί ένα αντικείμενο με αρχικό αποθηκευτικό χώρο size.

```
public SetOfStrings()
```

Δημιουργεί ένα σύνολο με αρχικό αποθηκευτικό χώρο ίσο με 10.

```
public SetOfStrings(SetOfStrings source)
```

Δημιουργεί ένα αντίγραφο του source

```
public boolean contains(String s)
```

Ελέγχει αν αυτό το σύνολο περιλαμβάνει την αλφαριθμητική σειρά s

```
public void add(String s)
```

Προσθέτει την αλφαριθμητική σειρά s σε αυτό το σύνολο. Αν η s υπάρχει ήδη, η add τερματίζει χωρίς να μεταβάλει αυτό το σύνολο. Αν η s δεν υπάρχει ήδη, ελέγχεται η χωρητικότητα, αν επαρκεί προστίθεται το s αλλιώς αυξάνεται η χωρητικότητα και στη συνέχεια προστίθεται το s.

```
public boolean remove(String s)
```

Διαγράφει τη σειρά s από αυτό το σύνολο. Επιστρέφει true αν η σειρά s βρέθηκε και διαγράφηκε από αυτό το σύνολο και false αν η σειρά δεν βρέθηκε.

```
public int size()
```

Επιστρέφει το μέγεθος αυτού του συνόλου, δηλαδή τον αριθμό των στοιχείων του

```
public String get(int i)
```

Επιστρέφει το στοιχείο που βρίσκεται στη θέση i αυτού του συνόλου

```
public SetOfStrings union(SetOfStrings sS)
```
Επιστρέφει την ένωση αυτού του συνόλου με το σύνολο sS

```
public SetOfStrings intersection(SetOfStrings sS)
```
Επιστρέφει την τομή αυτού του συνόλου με το sS

Να υλοποιηθεί η κλάση. Να προστεθούν όλες οι αναγκαίες μέθοδοι. Να προστεθεί main στην οποία να δημιουργείται ένα SetOfStrings με στοιχεία "0","1","2",…,"30" και ένα δεύτερο με στοιχεία "20","21","22",…,"50". Να εμφανιστεί η ένωση και η τομή των δύο συνόλων. Χρησιμοποιήστε για αποθηκευτικό χώρο πίνακα από Strings. Δημιουργήστε τον για ένα αρχικό μήκος. Αναπροσαρμόστε το μήκος αυτόματα όποτε ο χώρος δεν επαρκεί.

11.10.3 Η κλάση StackOfStrings

Η κλάση StackOfStrings αναπαριστά μία στοίβα από Strings. Ένα stack, όπως εξηγείται στην ενότητα 9.1, είναι μια δομή LIFO (Last In First Out). Η κλάση περιλαμβάνει τις ακόλουθες μεθόδους:

```
public boolean isEmpty()
```
Επιστρέφει true αν αυτό το stack είναι άδειο.

```
public void push(String data)
```
Εισάγει στην κορυφή του stack το String data.

```
public String pop()
```
Αφαιρεί το στοιχείο από την κορυφή του stack και το επιστρέφει.

Χρησιμοποιείστε για αποθηκευτικό χώρο πίνακα από Strings. Δημιουργήστε τον για ένα αρχικό μήκος. Αναπροσαρμόστε το μήκος αυτόματα, όποτε ο χώρος δεν επαρκεί.

11.10.4 Η κλάση QuadraticEquation

Να υλοποιηθεί η κλάση QuadraticEquation. Κάθε αντικείμενο της κλάσης αντιπροσωπεύει μια εξίσωση β' βαθμού. Η κλάση περιλαμβάνει τις ακόλουθες μεθόδους:

```
public QuadraticEquation (double a, double b, double g)
```
Δημιουργεί ένα αντικείμενο.

```
public QuadraticEquation (double[] triwnymo)
```
Δημιουργεί ένα αντικείμενο.

```
public double discriminant()
```
Επιστρέφει τη διακρίνουσα αυτής της εξίσωσης.

```
public double x1()
```
Επιστρέφει τη μια από τις ρίζες αυτής της εξίσωσης. Αν η διακρίνουσά της είναι μικρότερη από το 0, επιστρέφει Double.NaN.

```
public double x2()
```
Επιστρέφει τη μια από τις ρίζες αυτής της εξίσωσης. Αν η διακρίνουσά της είναι μικρότερη από το 0, επιστρέφει Double.NaN. Αν η διακρίνουσα είναι ίση με το 0, τότε οι δύο ρίζες είναι ίσες.

11.10.5 Η κλάση MyDate

Να υλοποιηθεί η κλάση MyDate που περιλαμβάνει τρεις μεταβλητές στιγμιότυπου ως εξής:

```
private int day;
private MyMonth month;
private int year;
```

Η day αναπαριστά την ημέρα ενός μήνα, η year το έτος και η month τον μήνα. Ενώ η day και η year είναι ακέραιοι, η month είναι απαριθμήσιμος τύπος.

Η κλάση διαθέτει επίσης ένα στατικό πεδίο format το οποίο είναι επίσης απαριθμήσιμος τύπος με δύο αντικείμενα, το GREEK και το ENGLISH που καθορίζουν την επιστροφή της toString ως εξής: Αν η format έχει την τιμή GREEK τότε η ημερομηνία επιστρέφεται ως dd/mm/yy διαφορετικά ως mm/dd/yy.

Η κλάση διαθέτει μια στατική μέθοδο isDisekto(int) που επιστρέφει true εφόσον καλείται με παράμετρο που αντιπροσωπεύει δίσεκτο έτος. Σημειώστε πως δίσεκτα είναι τα έτη που διαιρούνται ακριβώς με το 400 ή διαιρούνται με το 4 αλλά όχι με το 100. Επιπλέον, η κλάση διαθέτει μια μέθοδο στιγμιότυπου isDisekto που επιστρέφει true εφόσον κληθεί από αντικείμενο δίσεκτου έτους.

Προσθέστε main που χαρακτηρίζει κάθε έτος από το 2017 έως το 2100 ως δίσεκτο ή μη.

Βιβλιογραφία

[1] “Object (Java Platform SE 8).” https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object- (accessed Oct. 15, 2021).

[2] O. O. Adekunle, Word Stemming/Hashing Algorithm: For Mail Classification. Place of publication not identified: LAP LAMBERT Academic Publishing, 2011.

[3] J.-P. Aumasson, W. Meier, R. Phan, and L. Henzen, The Hash Function BLAKE. Berlin Heidelberg: Springer-Verlag, 2014. doi: 10.1007/978-3-662-44757-4.

Κεφάλαιο 12

Σύνοψη
Στην ενότητα αυτή παρουσιάζονται κλάσεις της βιβλιοθήκης της Java που χρησιμοποιούνται συχνά κατά την ανάπτυξη εφαρμογών. Πιο συγκεκριμένα παρουσιάζονται οι κλάσεις String και StringBuilser για διαχείριση συμβολοσειρών, η κλάση ArrayList τα αντικείμενα της οποίας είναι στην ουσία εξελιγμένοι πίνακες, η κλάση LinkedList που αναπαριστά διασυνδεδεμένες λίστες, η κλάση Stack που υποστηρίζει δομές LIFO, η PriorityQueue που υποστηρίζει ουρές προτεραιότητας, οι κλάσεις HashSet και TreeSet που υποστηρίζουν διαχείριση συνόλων, οι κλάσεις HashMap και TreeMap που υποστηρίζουν απεικονίσεις, δηλαδή συσχέτιση ενός κλειδιού με δεδομένα, η κλάση BigDecimal που υποστηρίζει διαχείριση πραγματικών αριθμών αντιμετωπίζοντας τα προβλήματα που πηγάζουν από τον προσεγγιστικό χαρακτήρα τους και τέλος, μια ομάδα κλάσεων που διευκολύνουν τη διαχείριση ημερομηνιών.

Προαπαιτούμενη γνώση
Το διαδικαστικό μοντέλο προγραμματισμού με Java και η Ενσωμάτωση.

Λέξεις κλειδιά
Λίστες (List), Ουρές (Queue), Ουρές προτεραιότητας (Priority Queue), κατακερματισμός (hashing), BigDecimal, Απεικονίσεις (Map), Δένδρα (Tree), Σύνολα (Set), Αμεταβλητότητα (immutability).

12 Αναγκαίες κλάσεις

Στην ενότητα αυτή παρουσιάζουμε βασικές κλάσεις της βιβλιοθήκης της Java που χρησιμοποιούνται συχνότατα και θα μας είναι χρήσιμες κατά την ανάπτυξη των ασκήσεων και άλλων εφαρμογών. Καταρχάς, η συζήτηση αφορά τις συμβολοσειρές, με την παρουσίαση των κλάσεων String και StringBuilder. Στη συνέχεια, εξετάζονται οι πιο βασικές συλλογές, δηλαδή κλάσεις κατάλληλες για την αποθήκευση αντικειμένων. Μετά παρουσιάζονται ορισμένες απεικονίσεις, δηλαδή κλάσεις με τις οποίες μπορούμε να συσχετίζουμε κλειδιά με δεδομένα, ακολουθούμενες από την κλάση BigDecimal για αναπαράσταση και διαχείριση πραγματικών τιμών και τέλος συζητάμε βασικές κλάσεις διαχείρισης ημερομηνιών.

Να σημειωθεί ότι δεν γίνεται εξαντλητική παρουσίαση των κλάσεων. Παρουσιάζεται η βασική λειτουργικότητά τους και οι πιο συχνά χρησιμοποιούμενες μέθοδοι και μόνον εφόσον αυτές είναι δυνατόν να κατανοηθούν με βάση τη γνώση που έχει παρουσιαστεί μέχρι εδώ σε αυτό το βιβλίο.

12.1 Η κλάση String

Κάποια στοιχεία από την κλάση String έχουν ήδη παρουσιαστεί. Εδώ θα παρουσιάσουμε μια σειρά από συναρτήσεις στιγμιότυπου και τους βασικούς δημιουργούς της κλάσης.

```
String s1 = new String("John");
String s2 = "John";
char[] p = {'M', 'a', 'r', 'y'};
String s3 = new String(p);
```

Κώδικας 12.1 *Οι βασικοί δημιουργοί της κλάσης String*

Στον κώδικα 12.1 βλέπουμε παραδείγματα των βασικών δημιουργών της κλάσης. Οι δύο πρώτοι δημιουργοί έχουν ήδη παρουσιαστεί (ενότητα 3.8). Στην τελευταία γραμμή βλέπουμε έναν δημιουργό που λαμβάνει ως παράμετρο έναν πίνακα από χαρακτήρες και συνθέτει ένα String.

Συναρτήσεις

```
char charAt  (int index)
```

επιστρέφει τον χαρακτήρα που βρίσκεται στη θέση index αυτού του String, δηλαδή του String που την καλεί.

```
int compareTo(String anotherString)
```

Επιστρέφει 0 αν αυτό το String είναι ίσο με το anotherString, αρνητικό ακέραιο αν αυτό το String είναι λεξικογραφικά μικρότερο από το anotherString και θετικό ακέραιο, αν αυτό το String είναι μεγαλύτερο. Για παράδειγμα, η πρόταση

```
System.out.println((s1 + " Black").compareTo(s2));
```

θα επιστρέψει θετικό ακέραιο. Πιο συγκεκριμένα, η αξιολόγηση της έκφρασης μέσα στην println γίνεται ως εξής: Πρώτα αξιολογείται η σύνδεση s1+" Black" που αποκτά προτεραιότητα καθώς περικλείεται σε παρένθεση. Το αποτέλεσμα αυτής της σύνδεσης είναι το String John Black. Το String αυτό συγκρινόμενο με την παράμετρο s2 που έχει περιεχόμενο John θεωρείται λεξικογραφικά μεγαλύτερο.

```
int compareToIgnoreCase(String anotherString)
```

Συγκρίνει αυτό το String με την παράμετρό της χωρίς να λαμβάνει υπόψη τυχόν διαφορές μεταξύ πεζών και κεφαλαίων. Για παράδειγμα, η κλήση

```
s1.compareToIgnoreCase("john");
```

επιστρέφει 0, παρότι το s1 έχει ως περιεχόμενο το John.

```
boolean endsWith(String suffix)
```

Επιστρέφει true εφόσον η παράμετρος suffix είναι το τελευταίο τμήμα αυτού του String. Αντίστοιχα, λειτουργεί και η startsWith.

```
boolean equalsIgnoreCase(String anotherString)
```

Επιστρέφει true εφόσον αυτό το String είναι ίσο με την παράμετρό της χωρίς να λαμβάνονται υπόψη διαφορές μεταξύ πεζών και κεφαλαίων. Η συνάρτηση μοιάζει πολύ με την compareToIgnoreCase με τη διαφορά ότι είναι boolean, ενώ η compareToIgnoreCase επιστρέφει int. Βεβαίως, όπως όλες οι κλάσεις της Java, η κλάση String υποστηρίζει μέθοδο equals.

```
int indexOf(String str)
```

Επιστρέφει τη θέση της str μέσα σε αυτό το String. Αν η str δεν βρεθεί, επιστρέφει -1.
Η συνάρτηση lastIndexOf(String str) επιστρέφει την τελευταία θέση της str μέσα σε αυτό το String. Αν η str δεν βρεθεί, επιστρέφει -1.

```
int indexOf(String str, int fromIndex)
```

Επιστρέφει τη θέση της str μέσα σε αυτό το String εφόσον αυτό βρεθεί από τη θέση fromIndex και μετά, διαφορετικά, επιστρέφει -1.

```
int length()
```

Επιστρέφει το μήκος αυτού του String.

```
boolean isEmpty()
```

Επιστρέφει true μόνον εφόσον το μήκος του String είναι ίσο με το 0.

```
boolean isBlank()
```

Επιστρέφει true αν το μήκος του String είναι 0 ή αν το String περιέχει μόνο λευκούς χαρακτήρες (whitespace characters). Σημειώστε πως λευκοί χαρακτήρες είναι όσοι χαρακτήρες ή σειρές χαρακτήρων αντιπροσωπεύουν τυπογραφικά οριζόντιο ή κάθετο διάστημα. Για παράδειγμα, ο κώδικας

```
System.out.println(s.length()+"                "+s.isEmpty()+"
"+s.isBlank());
```

θα τυπώσει

```
0 true true
```

εφόσον, s==”” και

```
1 false true
```

αν s==” ”;

```
String repeat(int count)
```

Επιστρέφει ένα String με τιμή την τιμή αυτού του String επαναλαμβανόμενη count φορές. Για παράδειγμα, η κλήση

```
System.out.println(s1.repeat(2));
```

θα εμφανίσει JohnJohn.

```
String replace (String s1, String s2)
```

Είναι υπερφορτωμένη ώστε να δέχεται ως παραμέτρους String ή char. Αντικαθιστά σε αυτό το String την πρώτη εμφάνιση της πρώτης παραμέτρου της με τη δεύτερη παράμετρο. Για παράδειγμα,

```
String s3 = "John came home yesterday";
System.out.println(s3.replace("John", "Mary"));
System.out.println(s3.replace('M', 'm'));
```

Η πρώτη κλήση της replace λαμβάνει τα Strings John και Mary και επιστρέφει ένα String ίδιο με το s3 στο οποίο το τμήμα John έχει αντικατασταθεί απο το Mary. Η δεύτερη κλήση λαμβάνει δύο χαρακτήρες και αντικαθιστά το M με το m. Προσοχή, καμία κλήση της replace δεν μεταβάλλει το s3. Σε όλες τις περιπτώσεις επιστρέφεται ένα νέο String. Όπως εξηγείται στην ενότητα 12.1.1, αυτό ισχύει για όλες τις συναρτήσεις της κλάσης String.

```
String[] split  (String regex)
```

Λαμβάνει ως παράμετρο ένα String που αναπαριστά μια κανονική έκφραση. Οι κανονικές εκφράσεις είναι έξω από την ύλη αυτού του βιβλίου. Μας είναι χρήσιμο όμως να γνωρίζουμε ότι ο χαρακτήρας διάστημα μπορεί να χρησιμοποιηθεί ως όρισμα στη συνάρτηση. Στην περίπτωση αυτή, η συνάρτηση επιστρέφει έναν πίνακα από Strings που συνιστούν τμήματα αυτού του String διαχωρισμένα με τον χαρακτήρα διάστημα. Για παράδειγμα, ο κώδικας

```
String s3 = "John came home yesterday";
String[] t = s3.split(" ");
System.out.println(Arrays.asList(t));
```

θα τυπώσει

```
[John, came, home, yesterday]
```

```
String stripLeading()
```

Διαγράφει όλους τους λευκούς χαρακτήρες από την αρχή αυτού του String. Για παράδειγμα, ο κώδικας

```
String s5 = "   John";
System.out.println(s5.stripLeading().equals("John"));
```

τυπώνει true. Αντίστοιχα λειτουργεί η stripTrailing που διαγράφει τους λευκούς χαρακτήρες από το τέλος αυτού του String. Η δε συνάρτηση trim() διαγράφει τους λευκούς χαρακτήρες από την αρχή και από το τέλος αυτού του String.

```
String substring(int beginIndex)
String substring(int beginIndex, int endIndex)
```

Επιστρέφει ένα τμήμα αυτού του String. Είναι υπερφορτωμένη. Το τμήμα που επιστρέφεται αρχίζει από την θέση beginIndex έως το τέλος του String ή έως και τη θέση endIndex-1, ανάλογα με τις πραγματικές παραμέτρους. Για παράδειγμα, ο κώδικας

```
System.out.println(s3.substring(5));
System.out.println(s3.substring(5, 10));
```

εμφανίζει

```
came home yesterday
came
```

Τέλος, οι συναρτήσεις toLowerCase() και toUpperCase() επιστρέφουν το String που τις καλεί με πεζά ή κεφαλαία, αντίστοιχα.

12.1.1 Αμεταβλητότητα

Τα αντικείμενα της κλάσης String είναι αμετάβλητα (immutable). Αυτό σημαίνει πως από τη στιγμή της δημιουργίας τους μέχρι το τέλος της εφαρμογής δεν είναι δυνατόν να μεταβληθεί το περιεχόμενό τους. Καμία από τις συναρτήσεις που συζητήσαμε παραπάνω δεν μεταβάλλει το περιεχόμενο ενός String. Για παράδειγμα, ο κώδικας

```
String s6 = new String("JOHN");
s6 = s6.toLowerCase();
```

έχει ως αποτέλεσμα να δημιουργηθούν τα Strings JOHN και john. Αρχικά, η μεταβλητή s6 αναφέρεται στο JOHN, στη συνέχεια όμως ανακατευθύνεται στο john και το JOHN μένει χωρίς αναφορά και επομένως είναι υποψήφιο για να συλλεγεί από τον συλλέκτη απορριμάτων.

Τα Strings είναι αμετάβλητα ώστε να είναι ασφαλή από την επίδραση διαφορετικών νημάτων (thread safe). Σε μια πολυνηματική εφαρμογή, διαφορετικά προγράμματα μοιράζονται τις ίδιες μεταβλητές. Έτσι είναι δυνατόν ένα πρόγραμμα να μεταβάλει την τιμή ενός String και να δημιουργήσει προβλήματα σε ένα άλλο πρόγραμμα που χρησιμοποιεί το ίδιο String.

Προσέξτε πως ο αμετάβλητος χαρακτήρας των String καθιστά τον σταδιακό υπολογισμό τους ιδιαίτερα κοστοβόρο. Για παράδειγμα, η συνάρτηση

```
static String buildString() {
     String rVal = "";
     for (char c = 'A'; c <= 'Z'; c++) {
          rVal += c;
     }
     return rVal;
}
```

δημιουργεί καταρχάς ένα κενό String. Στη συνέχεια μέσα στον βρόγχο for προσθετει έναν-έναν τους χαρακτήρες του αγγλικού αλφάβητου. Κάθε φορά που προστίθεται ένας χαρακτήρας δημιουργείται ένα νέο String, δηλαδή η συνάρτηση δημιουργεί ένα σύνολο από 27 συνολικά Strings ενώ υποτίθεται ότι χρειάζεται μόνο ένα.

12.2 Η κλάση StringBuilder

Τα αντικείμενα της κλάσης StringBuilder συνιστούν μεταβλητές συμβολοσειρές. Έτσι, όποτε χρειαζόμαστε να διαμορφώσουμε σταδιακά μια συμβολοσειρά, είναι προτιμότερο να χρησιμοποιούμε ένα αντικείμενο

StringBuilder αντί για String. Στη συνέχεια, αφού ολοκληρώσουμε τη διαμόρφωση της συμβολοσειράς μπορούμε να δημιουργήσουμε ένα αντικείμενο String.

```
static String buildString() {
    StringBuilder rVal = new StringBuilder(26);
    for (char c = 'A'; c <= 'Z'; c++) {
        rVal.append(c);
    }
    return new String(rVal);
}
```

Κώδικας 12.2 *Διαμόρφωση συμβολοσειράς με την StringBuilder*

Στον κώδικα 12.2 δημιουργούμε καταρχάς ένα αντικείμενο, rVal, τύπου StringBuilder με χωρητικότητα 26 χαρακτήρων. Στη συνέχεια προσθέτουμε έναν-έναν τους χαρακτήρες του αγγλικού αλφάβητου. Κάθε εκτέλεση της append τοποθετεί έναν χαρακτήρα στην πρώτη διαθέσιμη θέση του εσωτερικού αποθηκευτικού χώρου του rVal. Αφού η συμβολοσειρά φτάσει σε μια τελική μορφή, δημιουργούμε και επιστρέφουμε ένα String.

Στις περιπτώσεις που δεν γνωρίζουμε εκ των προτέρων το απαιτούμενο μήκος, μπορούμε να χρησιμοποιήσουμε τον δημιουργό χωρίς παραμέτρους της κλάσης StringBuilder. Ο δημιουργός χωρίς παραμέτρους κατασκευάζει ένα αντικείμενο αρχικής χωρητικότητας 16 χαρακτήρων. Έτσι κι αλλιώς, αν κατά την εκτέλεση του προγράμματος προκύψουν αυξημένες απαιτήσεις, ο εσωτερικός αποθηκευτικός χώρος αυξάνεται αυτόματα.

Ένας επιπλέον δημιουργός που μας ενδιαφέρει λαμβάνει ως παράμετρο ένα String με βάση το οποίο κατασκευάζει το αντικείμενο StringBuilder.

Η κλάση διαθέτει τις μεθόδους, charAt, compareTo, indexOf, lastIndexOf και substring που λειτουργούν παρόμοια με τις αντίστοιχες μεθόδους της κλάσης String. Έχει όμως και άλλες μεθόδους, κυριότερες από τις οποίες είναι:

```
int capacity()
```

Επιστρέφει το μέγεθος του εσωτερικού αποθηκευτικού χώρου αυτού του αντικειμένου

```
StringBuilder delete(int start, int end)
```

Διαγράφει τους χαρακτήρες από τη θέση start έως και τη θέση end-1 αυτού του αντικειμένου.

```
StringBuilder deleteCharAt(int index)
```

Διαγράφει τον χαρακτήρα στη θέση index αυτού του αντικειμένου.

```
StringBuilder  insert(int index, String str)
```

Εισάγει το str στη θέση index.

```
StringBuilder replace(int start, int end, String str)
```

Αντικαθιστά τους χαρακτήρες από τη θέση start έως την end-1 με το str.

```
StringBuilder reverse()
```

Αντιστρέφει τη σειρά των χαρακτήρων.

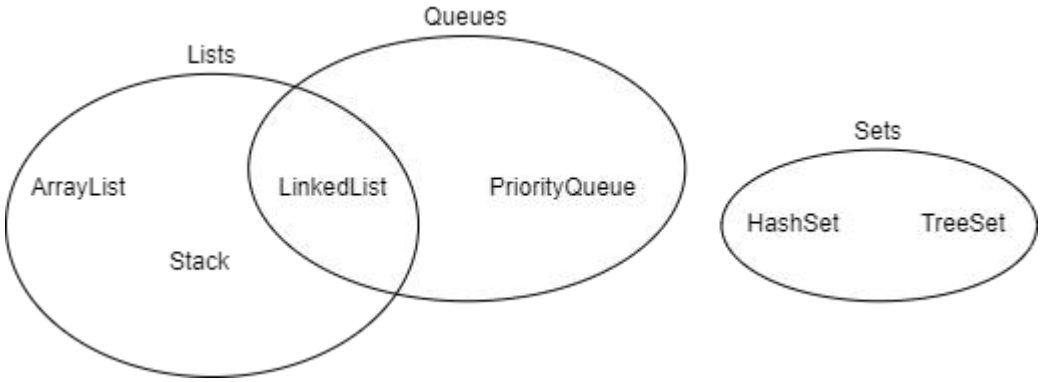
Προσοχή, η συνάρτηση equals της κλάσης δεν συγκρίνει δύο αντικείμενα ως προς το περιεχόμενό τους αλλά συγκρίνει τις διευθύνσεις τους, δηλαδή η equals επιστρέφει true μόνον εφόσον δύο μεταβλητές τύπου StringBuilder αναφέρονται στην ίδια θέση μνήμης.

Για περισσότερες λεπτομέρειες ανατρέξτε στην τεκμηρίωση της κλάσης.

12.3 Συλλογές

Σε αυτήν την ενότητα παρουσιάζουμε τις συλλογές (Collections), μια σημαντική κατηγορία κλάσεων της τυπικής βοβλιοθήκης της Java. Οι συλλογές είναι κλάσεις στα αντικείμενα των οποίων μπορούμε να

αποθηκεύουμε άλλα αντικείμενα. Οι συλλογές χωρίζονται σε τρεις κύριες υποομάδες, τις Λίστες (Lists), τις Ουρές (Queues) και τα Σύνολα (Sets).



Σχήμα 12.1 *Οι συλλογές της Java*

Πιο συγκεκριμένα, θα εξετάσουμε τις κλάσεις ArrayList, Stack, LinkedList, PriorityQueue, HashSet και TreeSet. Όπως φαίνεται στο σχήμα 12.1, η ArrayList, η Stack και η LinkedList ανήκουν στην ομάδα των Λιστών. Η δε LinkedList ανήκει και στην ομάδα των Ουρών όπου ανήκει και η PriorityQueue. Στα σύνολα ανήκουν οι κλάσεις HashSet και TreeSet.

Ένα ιδιαίτερο χαρακτηριστικό αυτών των κλάσεων είναι ότι συνιστούν παραμετροποιημένους τύπους ή γενικεύσεις (generics). Οι γενικεύσεις επιτρέπουν σε ένα τύπο ή μέθοδο να επενεργεί σε αντικείμενα διαφορετικών τύπων ενώ παρέχουν έλεγχο των τύπων κατά τον χρόνο μεταγλώττισης [1]. Τα αντικείμενα ενός γενικευμένου τύπου εξειδικεύονται κατά την δημιουργία τους. Για παράδειγμα, μπορεί κανείς να δημιουργήσει ένα ArrayList που αποθηκεύει αντικείμενα της κλάσης String ή ένα ArrayList που αποθηκεύει αντικείμενα μιας κλάσης οριζόμενης από τον χρήστη, π.χ. Person. Οι γενικεύσεις παρουσιάζονται αναλυτικά στην ενότητα 17. Εδώ όμως μπορούμε να δούμε πώς χρησιμοποιούνται τουλάχιστον σε ό,τι χρειάζεται για τη μελέτη των συλλογών.

Η κλάση Collections παρέχει μια σειρά από χρήσιμες στατικές μεθόδους διαχείρισης των συλλογών. Είναι για τις συλλογές κάτι αντίστοιχο με ότι είναι για τους πίνακες η κλάση Arrays. Δεν παρουσιάζεται εδώ αλλά αντίθετα όπου γίνεται χρήση της δίνονται οι απαραίτητες επεξηγήσεις. Φυσικά, ο ενδιαφερόμενος αναγνώστης μπορεί να ανατρέξει στην τεκμηρίωση της κλάσης για περισσότερες λεπτομέρειες.

12.3.1 Η κλάση ArrayList

Ένα αντικείμενο ArrayList συνιστά έναν εξελιγμένο πίνακα που μπορεί να αυξομειώσει το μέγεθός του όποτε προκύπτει ανάγκη, ενώ ταυτόχρονα προσφέρει διάφορες λειτουργίες που συχνά χρειαζόμαστε σε πίνακες. Όπως αναφέραμε προηγουμένως τα αντικείμενα ArrayList δημιουργούνται για την αποθήκευση συγκεκριμένου τύπου, π.χ.

```
ArrayList<String> l1 = new ArrayList<>();
ArrayList<Person> l2= new ArrayList<>();
```

Σε αυτόν τον κώδικα, δημιουργούνται τα ArrayList, l1 και l2. Το μεν l1 εξειδικεύεται ώστε να αποθηκεύει αντικείμενα τύπου String, το δε l2 εξειδικεύεται ώστε να αποθηκεύει αντικείμενα τύπου Person. Προσέξτε πως για τη δημιουργία αντικειμένων που αποτελούν εξειδικεύσεις γενικεύσεων απαραίτητη είναι η χρήση του τελεστή διαμαντιού (diamond operator) που συμβολίζεται με <>. Αν δεν χρησιμοποιηθεί ο τελεστής διαμαντιού, τότε δημιουργείται ένας πρωταρχικός τύπος (raw type). Για παράδειγμα, στην περίπτωση του ArrayList θα δημιουργηθεί ένα αντικείμενο σε κάθε θεση του οποίου είναι δυνατόν να αποθηκευτεί οποιοσδήποτε τύπος.

Η κλάση διατηρεί εσωτερικά έναν πίνακα όπου αποθηκεύει τα αντικείμενα. Παρότι ο πίνακας αυξομειώνεται αυτόματα όποτε χρειάζεται, παρέχεται η δυνατότητα να δημιουργηθεί με συγκεκριμένο μήκος έτσι ώστε όπου είναι εφικτό να καθίσταται ο κώδικας πιο αποτελεσματικός. Για παράδειγμα, ο κώδικας που ακολουθεί δημιουργεί ένα ArrayList εσωτερικής χωρητικότητας 30.

```
ArrayList<String> l3 = new ArrayList<>(30);
```

Σε περίπτωση που δεν οριστεί η χωρητικότητα κατά τη δημιουργία του πίνακα, τότε αυτή λαμβάνει την τιμή 10.
Αν χρειάζεστε έναν πίνακα από ArrayList αντικείμενα, τότε μπορείτε να τον δημιουργήσετε ως εξής.

```
ArrayList<String>[] t = new ArrayList[10];
```

Προσέξτε πως εδώ δημιουργείται κατάλληλος πίνακας για να αποθηκευτούν 10 αντικείμενα πρωταρχικού τύπου ArrayList, καθώς δεν είναι εφικτή η χρήση του τελεστή διαμαντιού κατά τη δημιουργία πινάκων γενικευμένων τύπων. Πάντως, κανένα από αυτά τα αντικείμενα δεν έχει δημιουργηθεί ακόμη. Έχει δημιουργηθεί μόνο ο πίνακας κάθε θέση του οποίου έχει τεθεί στο null. Στη συνέχεια, όμως, μπορείτε να δημιουργήσετε ένα αντικείμενο για κάθε θέση του πίνακα t, χρησιμοποιώντας τον τελεστή διαμαντιού όπως φαίνεται αμέσως από κάτω.

```
t[0]=new ArrayList<>(30);
```

Χρησιμοποιήστε ένα ArrayList εκεί που χρειάζεστε ταχεία πρόσβαση στα στοιχεία μιας λίστας. Ωστόσο, λάβετε υπόψη πως η πρόσθεση ή η διαγραφή στοιχείων από ενδιάμεση θέση ενός ArrayList είναι σχετικά χρονοβόρες διαδικασίες. Αν ένα στοιχείο θα πρέπει να διαγραφεί, π.χ., από το μέσον της λίστας, τότε ο εσωτερικός πίνακας θα πρέπει να μετακινήσει κατάλληλα όλα τα στοιχεία από το μέσον και μετά.

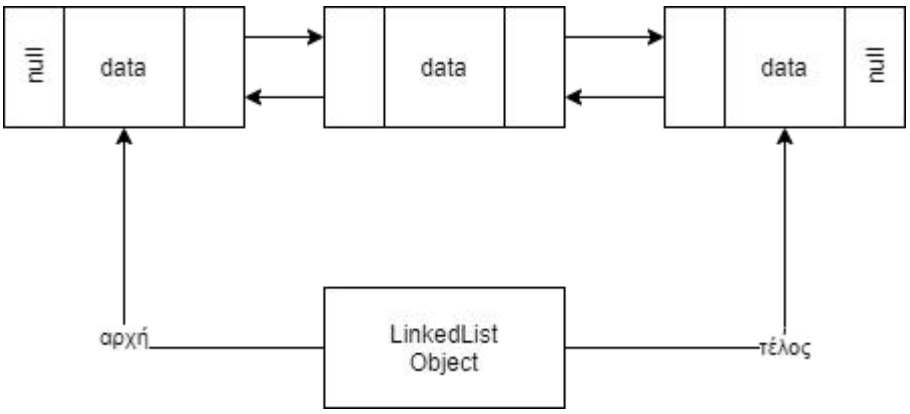
```
static void usage() {
    ArrayList<String> l = new ArrayList<>();
    l.add("1");
    l.add("2");
    l.add("4");
    l.add(2, "3");

    for (int idx = 0; idx < l.size(); idx++) {
        System.out.print(l.get(idx));
    }
    System.out.print("-");
    l.remove("2");
    System.out.println(l);
}
```

Κώδικας 12.3 *Παράδειγμα χρήσης της ArrayList*

Στον κώδικα 12.3, παρατίθεται παράδειγμα χρήσης της ArrayList. Καταρχάς, δημιουργούμε ένα ArrayList κατάλληλο για την αποθήκευση αντικειμένων της κλάσης String. Στη συνέχεια εισάγουμε τα Strings 1, 2 και 4. Στη συνέχεια, τοποθετούμε String με τιμή 3 στη θέση 2, δηλαδή ανάμεσα στις θέσεις των σειρών 2 και 4. Αμέσως μετά προσπελαύνουμε όλα τα στοιχεία της l με δεικτοδοτημένη for. Διαγράφουμε ένα στοιχείο της και την ξανατυπώνουμε. Οι κλήσεις l.add(2, "3") και l.remove("2") είναι οι πιο χρονοβόρες λειτουργίες στον κώδικα 12.3. Ο κώδικας εμφανίζει 1234-[1, 3, 4].

12.3.2 Η κλάση LinkedList

Τα αντικείμενα LinkedList συνιστούν αμφίδρομες λίστες. Επομένως, κάθε στοιχείο μίας τέτοιας λίστας, πλέον του αντικειμένου που αποθηκεύει, αποθηκεύει και τις διευθύνσεις του προηγούμενου και του επόμενου στοιχείου, όπως φαίνεται στο σχήμα 12.2.



Σχήμα 12.2 *Δομή της κλάσης LinkedList*

Σε μία τέτοια λίστα μπορεί κανείς να προσθέσει ή να αφαιρέσει στοιχεία ταχύτατα, καθώς οι λειτουργίες αυτές δεν απαιτούν μετακίνηση των στοιχείων. Αντίθετα, αργή είναι η διαδικασία προσπέλασης ενός συγκεκριμένου στοιχείου καθώς η αναζήτησή του αρχίζει αναγκαστικά από την αρχή ή το τέλος της λίστας και είναι σειριακή.

Όπως φαίνεται στο σχήμα 12.1, οι λίστες αυτές είναι ταυτοχρόνως και ουρές, δηλαδή μπορεί να χρησιμοποιηθούν ως First In – First Out (FIFO) δομές. Αυτό επιτυγχάνεται εύκολα όταν προσθέτουμε στοιχεία στο ένα άκρο της λίστας και τα αφαιρούμε από το άλλο. Επιπλέον, είναι προφανές ότι μπορούν να χρησιμοποιηθούν και σαν δομές LIFO.

```
static void usage() {
    LinkedList<String> l = new LinkedList<>();
    l.addFirst("1");
    l.addFirst("2");
    l.addLast("3");
    l.set(2, "4");
    System.out.print(l.peekFirst());
    System.out.print("-");
    while (!l.isEmpty()) {
        System.out.print(l.pollLast());
    }
}
```

Κώδικας 12.4 *Παράδειγμα χρήσης LinkedList*

Στον κώδικα 12.4, καταρχάς δημιουργούμε ένα αντικείμενο LinkedList κατάλληλο για να αποθηκεύουμε Strings. Στη συνέχεια προσθέτουμε στην αρχή της λίστας το String 1 και μετά το 2. Επομένως η λίστα έχει τα δεδομένα της στη σειρά 2-1. Στη συνέχεια, προσθέτουμε το 3 στο τέλος της λίστας, οπότε η λίστα γίνεται 2-1-3. Μετά στη θέση 2 της λίστας, δηλαδή εκεί που είναι αποθηκευμένο το 3, τοποθετούμε το 4 και η λίστα γίνεται 2-1-4. Στη συνέχεια, με την peekFirst ανακτούμε το στοιχείο που βρίσκεται στην αρχή της λίστας, δηλαδή το 2. Η peekFirst επιστρέφει το στοιχείο στην αρχή της λίστας χωρίς να το αφαιρεί από αυτήν.

Τέλος στην επαναληπτική διαδικασία, ενόσω η λίστα δεν είναι αδεια, αφαιρούμε και εμφανίζουμε το στοιχείο στην τελευταία θέση της λίστας. Η έξοδος της useExample είναι 2-412.

12.3.3 Η κλάση Stack

Η κλάση Stack παρέχει αντικείμενα που προσομοιώνουν τη δομή Last In – First Out (LIFO). Οτιδήποτε εισάγεται σε ένα Stack, εισάγεται στην κορυφή του και οτιδήποτε εξάγεται, λαμβάνεται από την κορυφή του. Επομένως, ένα τέτοιο αντικείμενο προσφέρει ταχύτατη πρόσβαση στην περίπτωση που χρησιμοποιείται όντως ως δομή LIFO. Αν όμως χρειαστεί να προσπελάσουμε αντικείμενα που δεν βρίσκονται στην κορυφή του Stack, τότε πρέπει να τα αναζητήσουμε σειριακά. Βεβαίως, είναι αυτονόητο ότι προσθέσεις ή αφαιρέσεις αντικειμένων σε άλλη θέση πέραν της κορυφής δεν έχουν νόημα σε ένα Stack.

```
static void usage() {
```

```
        Stack<String> s = new Stack<>();
        s.push("1");
        s.push("2");
        s.push("3");
        s.push("4");
        while (!s.isEmpty()) {
            System.out.println(s.pop());
        }
    }
```

Κώδικας 12.5 *Παράδειγμα χρήσης της Stack*

Στον κώδικα 12.5, ωθούμε με την push στη στοίβα s, τέσσερα String. Στη συνέχεια αφαιρούμε από τη στοίβα, ένα-ένα τα στοιχεία της έως ότου αδειάσει. Η έξοδος της συνάρτησης θα εμφανίσει 4321.

12.3.4 Η κλάση PriorityQueue

Η κλάση PriorityQueue προσομειώνει μια ουρά προτεραιότητας. Τα αντικείμενα που εισάγονται στην ουρά ταξινομούνται ως προς τη φυσική σειρά τους. Αν τα αντικείμενα δεν έχουν φυσική σειρά, τότε εφαρμόζονται άλλες τεχνικές που παρουσιάζονται στην ενότητα 14.1.2. Όταν ένα αντικείμενο εξάγεται από την ουρά, εξάγεται αυτό που έχει την μεγαλύτερη προτεραιότητα, δηλαδή τη μικρότερη σειρά στην ταξινομημένη ουρά. Στην περίπτωση που όλα τα αντικείμενα στην ουρά έχουν την ίδια προτεραιότητα, επιστρέφεται ένα αντικείμενο που επιλέγεται τυχαία.

```
    static void usage() {
        PriorityQueue<String> s = new PriorityQueue<>();
        s.add("John");
        s.add("Mark");
        s.add("Ann");
        s.add("Bill");
        while (!s.isEmpty()) {
            System.out.println(s.poll());
        }
    }
```

Κώδικας *12.6 Παράδειγμα χρήσης ουράς προτεραιότητας*

Στον κώδικα 12.6 εισάγουμε τέσσερα Strings στην ουρά προτεραιότητας s με τη συνάρτηση add. Στη συνέχεια τα ανακτούμε ένα προς ένα με την poll. Η έξοδος της useExample εμφανίζει τα Strings ταξινομημένα κατ' αύξουσα αλφαριθμητική σειρά. Σημειώστε πως τα αντικείμενα String έχουν φυσική σειρά.

12.3.5 Οι κλάσεις HashSet και TreeSet

Τρία είναι τα κύρια στοιχεία που χαρακτηρίζουν τα αντικείμενα της κλάσης HashSet. Κατά πρώτον είναι σύνολα (sets). Σε ένα σύνολο ένα στοιχείο μπορεί να υπάρχει μόνο μία φορά. Επομένως, αν επιχειρήσουμε να εισάγουμε σε ένα HashSet αντικείμενο που ήδη υπάρχει, το σύνολο παραμένει αμετάβλητο. Επιπλέον, η σειρά των στοιχείων ενός συνόλου είναι αυθαίρετη και δεν παραμένει σταθερή κατά τη διάρκεια του χρόνου. Τέλος, τα σύνολα αυτά χρησιμοποιούν τους κωδικούς κατακερματισμού των αντικειμένων που αποθηκεύουν ώστε να επιταχύνουν τις λειτουργίες της προσπέλασης, πρόσθεσης και διαγραφής.

```
    static void usage() { //Κώδικας 12.7
        HashSet<String> h = new HashSet<>();
        h.add("1");
        h.add("2");
        h.add("3");
        h.add("4");
        h.add("1");
        System.out.println(h);
        h.remove("2");
        if (h.contains("3")) {
```

```
            System.out.println("element found");
        }
        for (String s : h) {
            System.out.print(s);
        }
    }
```

Κώδικας 12.7 *Παράδειγμα χρήσης HashSet*

Στον κώδικα 12.7 παρατίθεται παράδειγμα χρήσης ενός HashSet. Όπως ήδη αναφέρθηκε, η πρόσθεση (add) στοιχείου και η διαγραφή (remove) είναι ταχείς διεργασίες καθώς χρησιμοποιείται ο κωδικός κατακερματισμού. Για τον ίδιο λόγο ταχύς είναι και ο έλεγχος αν το σύνολο περιέχει (contains) κάποιο στοιχείο. Τα στοιχεία αυτής της συλλογής, μπορούμε να τα προσπελάσουμε με την ενισχυμένη for, όπως δείχνει ο κώδικας 12.7. Σημειώστε πως δεν παρέχεται η δυνατότητα να ανακτήσουμε ένα στοιχείο από συγκεκριμένη θέση (get(i)), καθώς τα στοιχεία του συνόλου δεν διατηρούν σταθερή σειρά.

Παρόμοια με την κλάση HashSet είναι και η TreeSet, μόνο που δεν βασίζεται σε κωδικούς κατακερματισμού αλλά σε δυαδικά δένδρα αναζήτησης.

12.4 Οι κλάσεις HashMap και TreeMap

Οι κλάσεις HashMap και TreeMap ανήκουν σε ίδια ομάδα κλάσεων, στις λεγόμενες απεικονίσεις (Maps). Κάθε στοιχείο μιας απεικόνισης συσχετίζει δύο αντικείμενα, το κλειδί με την τιμή. Για παράδειγμα, έστω ότι θέλουμε να μετρήσουμε το πλήθος κάθε λέξης ενός κειμένου. Προφανώς, μας διευκολύνει μια δομή όπου η λέξη θα είναι κλειδί και το πλήθος η τιμή. Εύκολα μπορούμε να το πετύχουμε με ένα HashMap.

```
    static void usage(String text) {
        HashMap<String, Integer> hM = new HashMap<>();
        String[] words = text.split(" ");
        System.out.println(Arrays.asList(words));
        for (String word : words) {
            Integer cnt = hM.get(word);
            if (cnt==null) {
                cnt=0;
            }
            hM.put(word, cnt+1);
        }
        System.out.println(hM);
    }
```

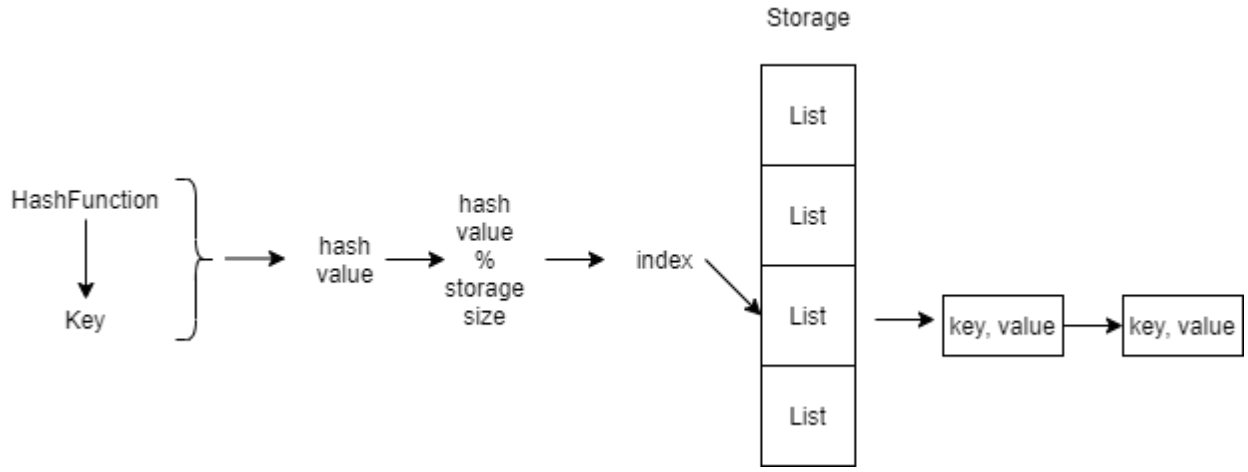
Κώδικας 12.8 *Μέτρηση των λέξεων κειμένου με την HashMap*

Στον κώδικα 12.8 παρουσιάζουμε ένα παράδειγμα όπου μετράμε το πλήθος κάθε λέξης του text με τη βοήθεια του HashMap, hM. Η κλάση HashMap είναι παραμετροποιημένη ως προς δύο τύπους. Ο πρώτος αναπαριστά το κλειδί (key) και ο δεύτερος την τιμή (value). Το αντικείμενο που έχουμε δημιουργήσει εδώ έχει ως κλειδί ένα String και ως τιμή έναν Integer. Κάθε εγγραφή στο hM περιλαμβάνει μια λέξη και το πλήθος που αυτή βρίσκεται στο text, δηλαδή είναι της μορφής <λέξη, πλήθος>. Προσέξτε όμως πως το πλήθος δεν θα μπορούσε να είναι τύπου int, καθώς οι τύποι των γενικεύσεων δεν μπορεί να είναι θεμελιώδεις τύποι. Μετά τη δημιουργία του hM, παίρνουμε τις λέξεις του κειμένου όπως αυτές διαχωρίζονται με διαστήματα στον πίνακα words. Για απλοποίηση, υποθέτουμε πως το κείμενο δεν περιλαμβάνει σημεία στίξης.

Στη συνέχεια, για κάθε λέξη word στον πίνακα words, ελέγχουμε με την get αν υπάρχει εγγραφή στο hM. Αν η get βρει εγγραφή μας επιστρέφει το πλήθος, αν όχι επιστρέφει null. Εφόσον δεν υπάρχει ήδη εγγραφή, αρχικοποιούμε τον μετρητή cnt στο 0. Στη συνέχεια, με την put ενημερώνουμε την εγγραφή. Προσέξτε πως αν εγγραφή με αυτό το κλειδί δεν υπάρχει ήδη, η put την εισάγει, διαφορετικά την ενημερώνει.

Χρήσιμο είναι να γνωρίζουμε τη βασική ιδέα στην οποία βασίζεται η κλάση HashMap.

Ο σκοπός της κλάσης HashMap είναι να παρέχει ταχεία πρόσβαση στα δεδομένα με βάση κάποιο κλειδί. Προς επίτευξη του σκοπού, αρχικά υπολογίζεται η τιμή hash του κλειδιού, όπως φαίνεται στο σχήμα 12.3. Ο υπολογισμός γίνεται από τη συνάρτηση hashCode του κλειδιού. Κάθε αντικείμενο HashMap διατηρεί έναν εσωτερικό πίνακα storage. Τα στοιχεία αυτού του πίνακα είναι συνδεδεμένες λίστες.



Σχήμα 12.3 *Η δομή HashMap*

Στη συνέχεια υπολογίζεται το ακέραιο υπόλοιπο της τιμής hash δια του μεγέθους του storage. Το αποτέλεσμα βρίσκεται υποχρεωτικά στο εύρος 0..storage.length, οπότε χρησιμοποιείται σαν δείκτης πρόσβασης στον storage. Από τη διαδικασία που περιγράφηκε μέχρι εδώ είναι προφανές πως διαφορετικά κλειδιά ενδέχεται να αντιστοιχιστούν στην ίδια θέση του storage. Σε κάθε θέση του storage όμως υπάρχει μια συνδεδεμένη λίστα. Κάθε στοιχείο της λίστας περιέχει κλειδί και τιμή. Η λίστα ανιχνεύεται προκειμένου να βρεθεί το συγκεκριμένο κλειδί, οπότε επιστρέφεται η τιμή. Η λειτουργικότητα της κλάσης TreeMap είναι παρόμοια με της HashMap. Ωστόσο, αξίζει να επισημάνουμε τις ακόλουθες διαφορές:

1. Η TreeMap βασίζεται σε δυαδικά δέντρα αναζήτησης και όχι σε συνάρτηση κατακερματισμού.
2. Η HashMap επιτρέπει ένα κλειδί με null τιμή ενώ η TreeMap όχι.
3. Η TreeMap διατηρεί τη φυσική σειρά των αντικειμένων που αποθηκεύει ενώ η HashMap όχι. Επομένως, αν η εφαρμογή μας απαιτεί πλοήγηση στα δεδομένα μιας απεικόνισης σύμφωνα με τη φυσική σειρά τους, η TreeMap είναι προτιμητέα.
4. Συνήθως η HashMap είναι ταχύτερη.
5. Η TreeMap έχει πιο πλούσια λειτουργικότητα από την HashMap, δηλαδή παρέχει περισσότερες συναρτήσεις για πλοήγηση και ανάκτηση των δεδομένων της.

12.5 Η κλάση BigDecimal

Όπως έχουμε επισημάνει στην ενότητα 3.2 αλλά και όπως επιδεικνύεται στον κώδικα 4.8, κατά τους υπολογισμούς με τους θεμελιώδεις τύπους float και double προκύπτουν αποκλίσεις από τα αναμενόμενα αποτελέσματα που οφείλονται στον προσεγγιστικό χαρακτήρα αυτών των τύπων.

Επομένως, οι τύποι αυτοί δεν είναι κατάλληλοι για υπολογισμούς που απαιτούν μεγάλη ακρίβεια. Για τέτοιου είδους υπολογισμούς προσφέρεται η κλάση BigDecimal. Πέραν της μεγάλης ακρίβειας, η κλάση προσφέρει και μεθόδους για στρογγυλοποίηση στα απαιτούμενα ψηφία. Ας σημειωθεί ότι τα αντικείμενα της κλάσης είναι αμετάβλητα (immutable). Η κλάση ανήκει στο πακέτο java.math.

```
static void usage() {
    BigDecimal d1 = new BigDecimal(1),
            d2 = new BigDecimal(1),
            d3 = new BigDecimal(0.00001);
    for (int j = 1; j <= 10; j++) {
        d1 = d1.add(d3);
    }
    d2 = d2.add(d3.multiply(new BigDecimal(10)));
    System.out.println(d1);
    System.out.println(d2);
    System.out.println(d1.equals(d2));
```

```
        MathContext mC = new MathContext(5);
        d1 = d1.round(mC);
        d2 = d2.round(mC);
        System.out.println(d1);
        System.out.println(d2);
        System.out.println(d1.equals(d2));
        d1=new BigDecimal(1.00016);
        System.out.println(d1.round(mC));
    }
```

Κώδικας 12.9 *Παράδειγμα χρήσης της BigDecimal*

Στον κώδικα 12.9 καταρχάς ορίζουμε τους BigDecimal d1, d2 και d3 με τιμές 1, 1 και 0.00001, αντίστοιχα. Στη συνέχεια, προσθέτουμε 10 φορές το d3 στο d1 και στο d2 προσθέτουμε το δεκαπλάσιο του d1. Αν τα d1, d2 και d3 ήταν float ή double, ο έλεγχος d1==d2 θα επέστρεφε false. Αντίθετα, με την BigDecimal, η σύγκριση d1.equals(d2) επιστρέφει true που είναι και η μαθηματικά ορθή τιμή. Στη συνέχεια, στρογγυλοποιούμε τα d1 και d2 χρησιμοποιώντας το αντικείμενο mC της κλάσης MathContext. Κατά την κατασκευή του mC, περνάμε ως παράμετρο τη σταθερά 5. Στη συνέχεια, η round δημιουργεί αντικείμενα με 5 ψηφία συνολικά (unscaled value). Στο παράδειγμά μας, η στρογγυλοποίηση γίνεται προς τον πλησιέστερο γείτονα (nearest neighbor). Επειδή τα 5 ψηφία ακολουθεί το 0, η στρογγυλοποίηση παράγει 1.0001. Αν όμως τα 5 ψηφία ακολουθούνται από ψηφίο μεγαλύτερο ή ίσο του 6, τότε η στρογγυλοποίηση παράγει 1.0002. Ωστόσο υπάρχει η δυνατότητα να χρησιμοποιηθεί διαφορετικό πρότυπο στρογγυλοποίησης (rounding mode). Η έξοδος του κώδικα 12.9 έχει ως εξής:

```
1.000100000000000000008180305391403130954586231382563710212707519531250
1.000100000000000000008180305391403130954586231382563710212707519531250
true
1.0001
1.0001
true
1.0002
```

12.6 Διαχείριση Ημερομηνίας

Στην ενότητα αυτή θα συζητήσουμε τις κλάσεις που μας διευκολύνουν στη διαχείριση ημερομηνιών. Πρόκειται για ένα σύνολο κλάσεων που ανήκουν στο πακέτο java.time. Πιο συγκεκριμένα, θα συζητήσουμε τις κλάσεις LocalDate, LocalTime, LocalDateTime, Period και Duration, ενώ θα δούμε και τον απαριθμήσιμο τύπο Month.

Ας σημειώσουμε ότι η αρχική βιβλιοθήκη διαχείρισης ημερομηνιών της Java είναι ενσωματωμένη στο πακέτο java.util. Ωστόσο η νέα βιβλιοθήκη, διαθέσιμη από την έκδοση 8 της Java, είναι πληρέστερη ώστε ο μόνος λόγος για να ασχοληθεί κανείς με την παλαιότερη είναι αν χρειάζεται να εργαστεί επάνω σε παλαιό κώδικα που την χρησιμοποιεί. Σε κάθε περίπτωση, εδώ παρουσιάζεται η νέα βιβλιοθήκη java.time.

Μια ιδιομορφία των κλάσεων που θα παρουσιάσουμε είναι ότι δεν παρέχουν δημόσιο δημιουργό. Αντ' αυτού παρέχουν κατάλληλες συναρτήσεις δημιουργίας αντικειμένων, τις αποκαλούμενες μεθόδους κατασκευής (factory methods) [2]. Αυτή η σχεδιαστική επιλογή υπαγορεύτηκε κυρίως από την υψηλή πολυπλοκότητα που απαιτεί η κλήση των δημιουργών αυτών των κλάσεων. Πράγματι, η δημιουργία ενός αντικειμένου LocalDate με χρήση δημιουργού απαιτεί σύνθετες ρυθμίσεις που έχουν να κάνουν με διαφορές μεταξύ γεωγραφικών τοποθεσιών (timezones), διαφορετικές γλώσσες, διαφορετικά ημερολόγια, κ.ά. Η χρήση κατασκευαστικών μεθόδων παρέχει μεγαλύτερη εκφραστική ισχύ με αποτέλεσμα να εξειδικεύονται και να απλοποιείται η κλήση τους. Για παράδειγμα, η κατασκευή ενός αντικειμένου που αναπαριστά τη σημερινή ημερομηνία γίνεται με απλή κλήση της συνάρτησης now() της κλάσης LocalDate.

Ένα άλλο κρίσιμο στοιχείο είναι πως τα αντικείμενα των κλάσεων αυτών είναι αμετάβλητα (immutables).

Πριν ξεκινήσουμε την περιγραφή των κλάσεων να αναφέρουμε δύο απαριθμήσιμους τύπους που παρέχει η βιβλιοθήκη. Ο τύπος DayOfWeek παρέχει σταθερές για κάθε ημέρα της εβδομάδος, από MONDAY έως SUNDAY και ο τύπος Month παρέχει σταθερές για κάθε μήνα του έτους, από JANUARY έως DECEMBER.

12.6.1 Η κλάση LocalDate

Ένα αντικείμενο LocalDate αναπαριστά μια ημερομηνία που συμπεριλαμβάνει ημέρα, μήνα και έτος, χωρίς να συμπεριλαμβάνει την ώρα.

```
static void usage() {//Κώδικας 12.10
    LocalDate today = LocalDate.now();
    LocalDate dec15_71 = LocalDate.of(1971, Month.DECEMBER, 15);
    LocalDate jan1_71 = LocalDate.of(1971, 1, 15);
    LocalDate tommorow = today.plusDays(1);
    LocalDate afterAWeek = today.plusWeeks(1);
    LocalDate afterAMonth = today.plusMonths(1);
    LocalDate afterAYear = today.plusYears(1);
    LocalDate yesterday = today.minusDays(1);
    LocalDate beforeAWeek = today.minusWeeks(1);
    LocalDate beforeAMonth = today.minusMonths(1);
    LocalDate beforeAYear = today.minusYears(1);
    LocalDate afterAMonthAndAWeek = today.plusMonths(1).plusWeeks(1);
}
```

Κώδικας 12.10 *Παράδειγμα χρήσης της LocalDate*

Όπως φαίνεται στον κώδικα 12.10, η δημιουργία ενός αντικειμένου LocalDate που αναπαριστά τη σημερινή ημερομηνία γίνεται με κλήση της στατικής μεθόδου now. Για να δημιουργήσουμε μια συγκεκριμένη ημερομηνία χρησιμοποιούμε τη στατική μέθοδο of. Στον κώδικα 12.10 δημιουργούνται δύο τέτοιες ημερομηνίες, η dec15_71 και η jan1_71. Στη θέση της δεύτερης παραμέτρου της of που αναπαριστά τον μήνα, μπορεί να χρησιμοποιηθεί int ή ο απαριθμήσιμος τύπος Month. Στη συνέχεια επιδεικνύονται οι μέθοδοι plus και minus με τις οποίες μπορούμε να προσθέτουμε ή να αφαιρούμε από μια ημερομηνία. Προσέξτε πως αυτές οι μέθοδοι δεν επηρεάζουν το αντικείμενο που τις καλεί αλλά επιστρέφουν ένα νέο αντικείμενο με το ανάλογο περιεχόμενο. Η κλάση διαθέτει αναγνώστες όπως οι getYear(), getMonth(), getDayOfWeek(), getDayOfMonth(), κλπ.

12.6.2 Η κλάση LocalTime

Η κλάση LocalTime αναπαριστά την ώρα, σε ώρες, λεπτά, δευτερόλεπτα και δισεκατομυριοστά του δευτερολέπτου.

```
static void usage() {
    LocalTime currentTime = LocalTime.now();
    LocalTime t12_50 = LocalTime.of(12, 50);
    LocalTime t12_50_43 = LocalTime.of(12, 50, 43);
    LocalTime t12_50_43_625 = LocalTime.of(12, 50, 43, 625);
    LocalTime after2Hours = LocalTime.now().plusHours(2);
    LocalTime after2Minutes = LocalTime.now().plusMinutes(2);
    LocalTime after2Seconds = LocalTime.now().plusSeconds(2);
    LocalTime after1MillionNanos = LocalTime.now().plusNanos(1_000_000);
    LocalTime beforeAnHour = LocalTime.now().minusHours(1);
    LocalTime before2minutes = LocalTime.now().minusMinutes(2);
    LocalTime before3Seconds = LocalTime.now().minusSeconds(3);
    LocalTime before10Nanos = LocalTime.now().minusNanos(10);
}
```

Κώδικας 12.11 *Παράδειγμα χρήσης της LocalTime*

Όπως φαίνεται στον κώδικα 12.11, η μέθοδος now χρησιμοποιείται για τη δημιουργία αντικειμένου που αντιπροσωπεύει την ώρα του συστήματος κατά τον χρόνο εκτέλεσης της μεθόδου. Με τη μέθοδο of μπορούμε να δημιουργήσουμε αντικείμενα που αναπαριστούν συγκεκριμένες ώρες. Κατά τη δημιουργία τέτοιων αντικειμένων πρέπει υποχρεωτικά να καθορίσουμε την ώρα και τα λεπτά και προαιρετικά, τα δευτερόλεπτα και τα νανοδευτερόλεπτα. Στη συνέχεια, γίνεται επίδειξη των μεθόδων με τις οποίες προσθέτουμε ή αφαιρούμε χρόνο από ένα αντικείμενο LocalTime. Και εδώ, όπως και στην LocalDate, το αντικείμενο που

καλεί τη μέθοδο παραμένει αμετάβλητο, ενώ επιστρέφεται ένα νέο αντικείμενο. Η κλάση παρέχει κατάλληλους αναγνώστες, όπως getHour(), getMinute(), getSecond() και getNano().

Ας σημειωθεί ότι παρέχεται η κλάση LocalDateTime η οποία συνδυάζει ημερομηνία και ώρα σε ένα αντικείμενο.

12.6.3 Οι κλάσεις Period και Duration

Η κλάση Period αναπαριστά χρονικές περιόδους στη μορφή έτη, μήνες, ημέρες.

```
    static void usage() {
        Period p = Period.of(0, 3, 10),
                p1 = Period.of(3, 1, 40);
        System.out.println(Period.ofWeeks(6));
        System.out.println(Period.ofDays(60));
        System.out.println(Period.ofMonths(3));
        System.out.println(Period.ofYears(2));

        LocalDate today = LocalDate.now();
        LocalDate myBirthday = LocalDate.of(1981, Month.DECEMBER, 15);
        Period myAge = Period.between(myBirthday, today);
        System.out.println(myAge);

        Period fiveMonths = Period.ofMonths(5);
        LocalDate afterFiveMonths = today.plus(fiveMonths);
        System.out.println(afterFiveMonths);
    }
```

Κώδικας 12.12 *Παράδειγμα χρήσης της Period*

Ο κώδικας 12.12, καταρχάς επιδεικνύει κάποιες από τις δυνατότητες δημιουργίας αντικειμένων τύπου Period. Στη συνέχεια, επιδεικνύει πώς μπορούμε να υπολογίσουμε τη χρονική περίοδο μεταξύ δύο αντικειμένων LocalDate με τη μέθοδο between και τέλος πώς μπορούμε να προσθέσουμε μια περίοδο σε μία ημερομηνία. Η έξοδος του κώδικα 12.12 είναι η εξής:

```
P42D
P60D
P3M
P2Y
P39Y9M15D
2022-02-28
```

Η κλάση Duration μοντελοποιεί μια χρονική διάρκεια σε δευτερόλεπτα ή/και νανοδευτερόλεπτα.

```
    static void usage() {
        Duration classDuration = Duration.ofMinutes(330);
        Duration breakTime=Duration.ofMinutes(4*10);
        Duration study=classDuration.minus(breakTime);
        System.out.println(study);

        LocalTime startLessons=LocalTime.of(8,0);
        LocalTime endLessons=startLessons.plus(classDuration);
        System.out.println(Duration.between(endLessons, startLessons));

System.out.println(classDuration.toHours()+":"+classDuration.toMinutesPart());
    }
```

Κώδικας 12.13 *Παραδείγματα χρήσης της Duration*

Στον κώδικα 12.13 καταρχάς δημιουργούμε τη χρονική διάρκεια classDuration 330 λεπτών και την breakTime διάρκειας 40 λεπτών. Στη συνέχεια υπολογίζουμε και τυπώνουμε τη διάρκεια του μαθήματος.

Αμέσως μετά καθορίζουμε την ώρα έναρξης του μαθήματος και υπολογίζουμε την ώρα λήξης του. Τέλος, εμφανίζουμε την classDuration στη μορφή hours:minutes.
Η έξοδος του κώδικα 12.13 έχει ως εξής:

```
PT4H50M
PT-5H-30M
5:30
```

12.6.4 Διαμόρφωση

Συχνά προκύπτει η ανάγκη διαμόρφωσης των αντικειμένων χρόνου. Η τυπική διαμόρφωση των ημερομηνιών επιτυγχάνεται με χρήση της κλάσης DateTimeFormatter.

```
    static void formatDate() {
        DateTimeFormatter shortFormat =
DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
        DateTimeFormatter mediumFormat =
DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
        DateTimeFormatter longFormat =
DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG);
        DateTimeFormatter fullFormat =
DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
        LocalDate d = LocalDate.of(2021, 9,30);
        System.out.println(shortFormat.format(d));
        System.out.println(d.format(mediumFormat));
        System.out.println(d.format(longFormat));
        System.out.println(d.format(fullFormat));
    }
```

Κώδικας 12.14 *Τυπική διαμόρφωση ημερομηνίας*

Στον κώδικα 12.14 έχουμε δημιουργήσει τέσσερις διαμορφωτές, τους shortFormat, mediumFormat, longFormat και fullFormat. Στη συνέχεια εμφανίζουμε τη σημερινή ημερομηνία αξιοποιώντας τη μέθοδο format της κλάσης LocalTime ή εναλλακτικά, τη μέθοδο format της κλάσης DateTimeFormatter. Η έξοδος του κώδικα έχει ως εξής:

```
30/9/21
30 Σεπ 2021
30 Σεπτεμβρίου 2021
Πέμπτη, 30 Σεπτεμβρίου 2021
```

Η τυπική διαμόρφωση της ώρας επιτυγχάνεται επίσης με χρήση της DateTimeFormatter.

```
    static void formatTime() {//Κώδικας 12.15
        DateTimeFormatter shortFormat =
DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
        DateTimeFormatter mediumFormat =
DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
        LocalTime t = LocalTime.of(12, 50, 43);
        System.out.println(shortFormat.format(t));
        System.out.println(t.format(mediumFormat));
    }
```

Κώδικας 12.15 *Τυπική διαμόρφωση ώρας*

Ο κώδικας 12.15 παρέχει παραδείγματα τυπικής διαμόρφωσης της ώρας. Προσοχή, οι διαμορφωτές FormatStyle.LONG και FormatStyle.FULL δεν ισχύουν για τη διαμόρφωση της ώρας. Η έξοδος του κώδικα 12.15 έχει ως εξής:

```
12:50 μ.μ.
12:50:43 μ.μ.
```

Πέραν της τυπικής διαμόρφωσης, είναι δυνατόν να καθορίσουμε τη διαμόρφωση με βάση κάποιο μοτίβο με τη βοήθεια της συνάρτησης ofPattern.

```
static void customFormat() {
    LocalDate d = LocalDate.now();
    DateTimeFormatter f1 = DateTimeFormatter.ofPattern("MMMM, dd, yyyy");
    DateTimeFormatter f2 = DateTimeFormatter.ofPattern("dd/MM/yyyy");
    DateTimeFormatter f3 = DateTimeFormatter.ofPattern("dd-MMMM-yy");
    System.out.println(d.format(f2));
    System.out.println(d.format(f3));

    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("hh:mm:ss.nn a");
    LocalTime t = LocalTime.of(15,20,30,450);
    System.out.println(t.format(dtf));
    dtf = DateTimeFormatter.ofPattern("hh:mm a");
    System.out.println(t.format(dtf));
}
```

Κώδικας 12.16 *Διαμόρφωση ημερομηνίας και ώρας με βάση μοτίβο*

Ο κώδικας 12.16 παρουσιάζει παραδείγματα διαμόρφωσης ημερομηνιών και ωρών. Εύκολα μπορεί να καταλάβει κανείς από τον κώδικα πως ο χαρακτήρας y αντιστοιχεί σε ψηφίο έτους, ο M σε ψηφίο μήνα, ο d σε ψηφίο ημέρας, ο h σε ψηφίο ώρας, ο m σε ψηφίο λεπτού, ο s σε ψηφίο δευτερολέπτου και ο n σε ψηφίο νανοδευτερολέπτου. Τέλος, ο χαρακτήρας a έχει ως αποτέλεσμα η ώρα να συνοδεύεται από τη σύντμηση π.μ. ή μ.μ., ανάλογα με την τιμή της ώρας.
Η έξοδος του κώδικα 12.16 έχει ως εξής:

```
30/09/2021
30-Σεπτεμβρίου-21
03:20:30.450 μ.μ.
03:20 μ.μ.
```

Η κλάση DateTimeFormatter μπορεί να αξιοποιηθεί και κατά τη δημιουργία αντικειμένων ημερομηνίας και ώρας.

```
static void parsing() { //Κώδικας 12.17
    DateTimeFormatter fd = DateTimeFormatter.ofPattern("dd/MM/yyyy");
    DateTimeFormatter ft = DateTimeFormatter.ofPattern("hh:mm:ss");
    LocalDate d = LocalDate.parse("15/01/1987", fd);
    LocalTime t = LocalTime.parse("12:35:31", ft);
    System.out.println(d);
    System.out.println(t);
}
```

Κώδικας 12.17 *Δημιουργία ημερομηνίας και ώρας με βάση μοτίβο*

Στον κώδικα 12.17 έχουμε δημιουργήσει δύο διαμορφωτές, τον fd και τον ft, για ημερομηνίες και ώρες, αντίστοιχα. Στη συνέχεια, αξιοποιώντας τους διαμορφωτές, δημιουργούμε αντικείμενα LocalDate και LocalTime περνώντας στην parse τις τιμές ημερομηνίας ή ώρας σαν συμβολοσειρές που ακολουθούν το μοτίβο του αντίστοιχου διαμορφωτή.

12.7 Λυμένες Ασκήσεις

12.7.1 Η κλάση τράπουλα

Αντικαταστήστε τον πίνακα deck της κλάσης Deck (άσκηση 11.9.7) με κατάλληλο ArrayList.

Λύση

```
import java.util.ArrayList;
import java.util.Collections;

public class Deck {

    private final ArrayList<Card> deck;

    public Deck() {
        deck = new ArrayList<>(52);
        for (CardRank r : CardRank.values()) {
            for (Suit s : Suit.values()) {
                deck.add(new Card(r, s));
            }
        }
    }

    public void shuffle() {
        Collections.shuffle(deck);
    }

    public boolean empty() {
        return deck.isEmpty();
    }

    public Card pickCard() {
        return deck.remove(0);
    }

    @Override
    public String toString() {
        return deck.toString();
    }

    public static void main(String[] args) {
        Deck d1 = new Deck();
        Deck d2 = new Deck();
         for (int i = 0; i < 24; i++) {
            d1.pickCard();
        }
        System.out.println(d1);
        System.out.println("");
        d2.shuffle();
        System.out.println(d2);
    }
}
```

Κώδικας 12.18 *Η κλάση Deck με ArrayList*

Όπως φαίνεται στον κώδικα 12.18, η αντικατάσταση του πίνακα με ArrayList, απλοποιεί σημαντικά τον κώδικα. Η ιδιωτική μεταβλητή cardsInDeck δεν είναι πλέον απαραίτητη. Για το ανακάτεμα της τράπουλας δεν είναι αναγκαίο να αναπτύξουμε δικό μας αλγόριθμο, καθώς είναι διαθέσιμη κατάλληλη συνάρτηση από την κλάση Collections. Επίσης, η συνάρτηση toString απλοποιείται σημαντικά.

12.7.2 Δεύτερη έκδοση της κλάσης Person

Να υλοποιηθεί νέα έκδοση της κλάσης Person. Στη νέα έκδοση, κάθε αντικείμενο Person διαθέτει αριθμό ταυτότητας, μικρό όνομα, επώνυμο και ημερομηνία γέννησης. Ο αριθμός ταυτότητας είναι μια συμβολοσειρά που προσδιορίζει μονοσήμαντα ένα πρόσωπο, δηλαδή θεωρούμε πως δύο στιγμιότυπα με τον ίδιο αριθμό ταυτότητας αφορούν το ίδιο πρόσωπο. Επομένως, μετά τη δημιουργία ενός αντικειμένου Person, ο αριθμός ταυτότητας δεν μπορεί να μεταβληθεί. Η κλάση ορίζει όλες τις αναγκαίες μεθόδους και επιπλέον μια μέθοδο που επιστρέφει την ηλικία του προσώπου σε ακέραιο αριθμό ετών. Προσθέστε μέθοδο main στην οποία να δημιουργήσετε δύο αντικείμενα Person και να εμφανίσετε τις ηλικίες τους.

Λύση
Παραθέτουμε πρώτα τον κώδικα της κλάσης Person και μετά επεξηγήσεις.

```
import java.time.LocalDate;
import java.time.Month;
import java.time.Period;

public class Person {

    private final String id;
    private String fName;
    private String sName;
    private LocalDate birthday;

    public Person(String id, String fName, String sName, LocalDate birthday) {
        this.id = id;
        this.fName = fName;
        this.sName = sName;
        this.birthday = birthday;
    }

    public LocalDate getBirthday() {
        return birthday;
    }

    public void setBirthday(LocalDate birthday) {
        this.birthday = birthday;
    }

    public String getfName() {
        return fName;
    }

    public void setfName(String fName) {
        this.fName = fName;
    }

    public String getsName() {
        return sName;
    }

    public void setsName(String sName) {
        this.sName = sName;
    }

    public int getAge() {
        LocalDate today = LocalDate.now();
        return Period.between(birthday, today).getYears();
    }

    public String getId() {
        return id;
```

```
    }

    @Override
    public int hashCode() {
        int hash = 5;
        return 17 * hash + this.id.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Person other = (Person) obj;
        return this.id.equals(other.id);
    }

    @Override
    public String toString() {
        return fName + " " + sName + ", id=" + id;
    }

    public static void main(String[] args) {
        Person jim = new Person("A89132", "Jim", "Black", LocalDate.of(1971,
Month.MARCH, 5));
        Person john = new Person("A89132", "John", "Black", LocalDate.of(1972,
Month.APRIL, 5));
        System.out.println(jim + " is " + jim.getAge() + " years old ");
        System.out.println(john + " is " + john.getAge() + " years old ");
    }
}
```

Κώδικας 12.19 *Η δεύτερη έκδοση της κλάσης Person*

Ο δημιουργός της κλάσης Person, οι ρυθμιστές και οι αναγνώστες συνιστούν τυπικές υλοποιήσεις. Επειδή ο αριθμός ταυτότητας ταυτοποιεί ένα πρόσωπο είναι ιδιωτικός και final και επομένως δεν υπάρχει ρυθμιστής για τον αριθμό ταυτότητας, δηλαδή δεν μπορεί να μεταβληθεί μετά τη δημιουργία αντικειμένου Person.

Από τη στιγμή που θεωρούμε πως δύο αντικείμενα Person με τον ίδιο αριθμό ταυτότητας αφορούν το ίδιο πρόσωπο, οι συναρτήσεις equals και hashCode μπορούν να βασιστούν αποκλειστικά στο id.

Η συνάρτηση getAge βασίζεται στο παράδειγμα χρήσης της Period (κώδικας 12.12) με τη διαφορά ότι εκφράζει την ηλικία του προσώπου μόνον σε έτη.

Στην main δημιουργούμε δύο πρόσωπα και εμφανίζουμε τις ηλικίες τους σύμφωνα με την απαίτηση της εκφώνησης.

12.7.3 Η κλάση MyHashMap

Να αναπτυχθεί η κλάση MyHashMap που συσχετίζει ένα αντικείμενο Person (κώδικας 12.19) με το id του. Η κλάση να περιλαμβάνει τέσσερις μόνο μεθόδους και πιο συγκεκριμένα τις:

```
public Person put(Person person)
```

Εισάγει στο αντικείμενο MyHashMap, ένα αντικείμενο Person. Αν αντικείμενο Person με το ίδιο id δεν υπάρχει ήδη στο MyHashMap, η put το εισάγει και επιστρέφει null. Αν αντικείμενο Person με το ίδιο id βρίσκεται ήδη στο αντικείμενο MyHashMap, η put το επιστρέφει και ενημερώνει το MyHashMap με τη νέα τιμή.

`Person get(String id)`
Επιστρέφει το αντικείμενο Person με αριθμό ταυτότητας id ή null αν δεν βρεθεί το id.

`Person remove(String id)`
Αφαιρεί και επιστρέφει το αντικείμενο Person με αριθμό ταυτότητας id. Αν δεν βρεθεί αυτό το id επιστρέφει null.

`toString()`.
Επιστρέφει την String αναπαράσταση των αντικειμένων Person που περιέχονται στο MyHashMap, ένα ανα γραμμή.
Προσθέστε συνάρτηση main στην οποία

1. Δημιουργήστε ένα αντικείμενο MyHashMap
2. Εισάγετε 6 αντικείμενα Person
3. Διαβάστε με την get, από το αντικείμενο MyHashMap, ένα-ένα τα πρόσωπα που έχετε εισάγει και τυπώστε τα, ένα ανά γραμμή.
4. Μεταβάλλετε το μικρό όνομα ενός από τα 6 αντικείμενα Person και ενημερώστε το αντικείμενο MyHashMap. Διαβάστε από το MyHashMap το πρόσωπο του οποίου μεταβλήθηκε το όνομα και τυπώστε το ώστε να ελέγξετε πως η ενημέρωση έγινε σωστά.
5. Διαγράψτε ένα πρόσωπο από το αντικείμενο MyHashMap και τυπώστε το αντικείμενο ώστε να ελέγξετε πως η μέθοδος toString() δουλεύει σύμφωνα με την εκφώνηση.
6. Η κλάση δεν επιτρέπει την καταχώρηση αντικειμένων με null τιμή. Η άσκηση να λυθεί με τη λογική που παρουσιάζει το σχήμα 12.3, χωρίς να χρησιμοποιηθεί η κλάση HashMap.

Λύση

```
import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;

public class MyHashMap {

    private static final int SIZE = 3;
    private final ArrayList<Person>[] store = new ArrayList[SIZE];

    public Person put(Person person) {
        if (person == null) {
            return null;
        }
        String id = person.getId();
        int hashCode = id.hashCode();
        int idx = hashCode % store.length;
        if (store[idx] == null) {
            store[idx] = new ArrayList<>();
        }
        int idLocation = -1;
        for (int i = 0; i < store[idx].size(); i++) {
            if (store[idx].get(i).getId().equals(id)) {
                idLocation = i;
                break;
            }
        }

        Person oldValue;
        if (idLocation == -1) {
            store[idx].add(person);
            return null;
```

```
        } else {
            oldValue = store[idx].get(idLocation);
            store[idx].set(idLocation, person);
        }
        return oldValue;
    }

    Person get(String id) {
        int hashCode = id.hashCode();
        int idx = hashCode % store.length;
        if (store[idx] == null) {
            return null;
        }
        int idLocation = -1;
        for (int i = 0; i < store[idx].size(); i++) {
            if (store[idx].get(i).getId().equals(id)) {
                idLocation = i;
                break;
            }
        }
        if (idLocation == -1) {
            return null;
        }
        return store[idx].get(idLocation);
    }

    Person remove(String id) {
        int hashCode = id.hashCode();
        int idx = hashCode % store.length;
        if (store[idx] == null) {
            return null;
        }
        int idLocation = -1;
        for (int i = 0; i < store[idx].size(); i++) {
            if (store[idx].get(i).getId().equals(id)) {
                idLocation = i;
                break;
            }
        }
        if (idLocation == -1) {
            return null;
        }
        Person rVal = store[idx].get(idLocation);
        store[idx].remove(idLocation);
        return rVal;
    }

    @Override
    public String toString() {
        StringBuilder rVal = new StringBuilder("");
        for (ArrayList<Person> cStore : store) {
            if (cStore != null) {
                for (int j = 0; j < cStore.size(); j++) {
                    rVal.append(cStore.get(j)).append("\n");
                }
            }
        }
        return rVal.toString();
    }

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

```
        Person p1 = new Person("A1", "Jim", "Black", LocalDate.of(1961,
Month.MARCH, 1));
        Person p2 = new Person("A2", "Jack", "Blue", LocalDate.of(1967,
Month.MARCH, 1));
        Person p3 = new Person("A3", "Jim", "Green", LocalDate.of(1971,
Month.MARCH, 1));
        Person p4 = new Person("A4", "Mary", "White", LocalDate.of(1961,
Month.MARCH, 1));
        Person p5 = new Person("A5", "Ann", "Red", LocalDate.of(1967, Month.MARCH,
1));
        Person p6 = new Person("A6", "Kelly", "Pink", LocalDate.of(1971,
Month.MARCH, 1));
        MyHashMap map = new MyHashMap();
        map.put(p1);
        map.put(p2);
        map.put(p3);
        map.put(p4);
        map.put(p5);
        map.put(p6);

        System.out.println(map.get(p1.getId()));
        System.out.println(map.get(p2.getId()));
        System.out.println(map.get(p3.getId()));
        System.out.println(map.get(p4.getId()));
        System.out.println(map.get(p5.getId()));
        System.out.println(map.get(p6.getId()));

        p1.setfName("Jameson");
        map.put(p1);
        System.out.println("after fName replacement");
        System.out.println(map.get(p1.getId()));

        System.out.println(map.remove(p1.getId()));
        System.out.println("after remove One");
        System.out.println(map.get(p1.getId()));
        System.out.println("print map");
        System.out.println(map);
    }
}
```

Κώδικας 12.20 *Η κλάση MyHashMap*

Το πιο κρίσιμο σημείο για τη λύση αυτής της άσκησης είναι η επιλογή της κατάλληλης δομής. Στη λύση που προτείνεται στον κώδικα 12.20, έχει επιλεγεί ένας πίνακας από ArrayList. Κάθε ένα ArrayList περιέχει αντικείμενα Person. Η put λαμβάνει ως παράμετρο ένα Person. Στην τυπική περίπτωση, η put θα λάμβανε δύο παραμέτρους, ένα String που θα αντιπροσώπευε το id και θα ήταν το κλειδί της απεικόνισης και ένα Person που θα συσχετιζόταν με το id. Στη συγκεκριμένη εφαρμογή όμως γνωρίζουμε πως κάθε Person συσχετίζεται αποκλειστικά με το id του. Επομένως, το κλειδί της απεικόνισης είναι το id το οποίο μπορούμε να πάρουμε από την παράμετρο Person της put.

Στην περίπτωση που η put κληθεί με παράμετρο null απλώς επιστρέφει. Στη συνέχεια παράγει την τιμή κατακερματισμού και εντοπίζει τη θέση στον πίνακα store υπολογίζοντας το ακέραιο υπόλοιπο της τιμής κατακερματισμού με το μέγεθος του store. Αν στη θέση που αντιστοιχεί το id δεν υπάρχει λίστα, τότε αυτή δημιουργείται. Στη συνέχεια εισάγεται το αντικείμενο Person εφόσον αυτό δεν υπάρχει ήδη στην αντίστοιχη λίστα.

Η get λαμβάνει ως παράμετρο ένα id. Υπολογίζει σε ποια θέση του πίνακα αντιστοιχεί αυτό το id με παρόμοιο τρόπο με την put. Αν στη θέση εκείνη δεν υπάρχει λίστα, είναι προφανές πως το κλειδί και το αντίστοιχο πρόσωπο δεν έχουν καταχωρηθεί, οπότε επιστρέφει null. Αν βρεθεί λίστα, ανιχνεύεται σειριακά για το id. Αν βρεθεί επιστρέφεται το αντικείμενο Person, διαφορετικά null.

Αντίστοιχα λειτουργεί η remove, με τη διαφορά πως αν βρεθεί το αντικείμενο, διαγράφεται από την αντίστοιχη λίστα.

Η toString χρησιμοποιεί StringBuilder για να διαμορφώσει την τιμή επιστροφής της. Αφού τη διαμορφώσει την επιστρέφει σε μορφή String. Προσθέτει στην rVal, κάθε Person, κάθε λίστας του store. Μετά από κάθε Person προσθέτει τον χαρακτήρα '\n' ώστε να αλλάζει γραμμή σύμφωνα με τις απαιτήσεις της εκφώνησης.

Η main ακολουθεί απλώς τις οδηγίες της εκφώνησης. Προσέξτε το σημείο κατά το οποίο μεταβάλλουμε το μικρό όνομα ενός προσώπου, δηλαδή τον κώδικα

```
p1.setfName("Jameson");
map.put(p1);
System.out.println("after fName replacement");
System.out.println(map.get(p1.getId()));
```

Τι θα συμβεί αν διαγράψουμε την πρόταση map.put(p1). Κάντε μια υπόθεση. Προχωρήστε στη διαγραφή και τρέξτε πάλι την κλάση. Μαντέψατε σωστά; Αν όχι, εξηγήστε αυτό το αποτέλεσμα.

Τέλος, ας σημειωθεί πως στον κώδικα 12.20 ο πίνακας store δημιουργήθηκε με μέγεθος μόλις 3 για λόγους διευκόλυνσης του ελέγχου του κώδικα. Με αυτό το μέγεθος ακόμη και λίγα αντικείμενα Person αντιστοιχίζονται εύκολα στην ίδια θέση του πίνακα και επομένως προστίθενται περισσότερα από 1 αντικείμενα στην αντίστοιχη λίστα.

12.7.4 Η κλάση MyPriorityQueue

Σε ένα νοσοκομείο, οι ασθενείς ταξινομούνται σε τρεις σειρές προτεραιότητας, ασθενείς με χαμηλή, με μεσαία και με υψηλή προτεραιότητα. Οι ασθενείς εξυπηρετούνται ανάλογα με την προτεραιότητά τους. Μεταξύ ασθενών με την ίδια προτεραιότητα, εξυπηρετούνται πρώτα αυτοί που εισάγονται πρώτοι. Να υλοποιηθεί η κλάση MyPriorityQueue που υποστηρίζει μια ουρά προτεραιότητας κατάλληλη για τις ανάγκες εξυπηρέτησης των ασθενών του νοσοκομείου. Υλοποιήστε δύο βασικές μεθόδους, την add με την οποία εισάγεται ένας ασθενής στο σύστημα και την get η οποία επιστρέφει έναν ασθενή σύμφωνα με την κατάλληλη σειρά και παράλληλα τον διαγράφει από την ουρά. Για ασθενείς, χρησιμοποιήστε αντικείμενα της κλάσης Person (κώδικας 12.19). Προσθέστε main στην οποία δημιουργήστε ένα αντικείμενο MyPriorityQueue, εισάγεται έξι πρόσωπα, δύο με χαμηλή προτεραιότητα, δύο με μεσαία και δύο με υψηλή. Εξάγετε από την ουρά όλα τα πρόσωπα. Εμφανίστε τα και ελέγξτε ότι εξάγονται με τη σωστή σειρά.

Λύση

```
import java.time.LocalDate;
import java.time.Month;
import java.util.LinkedList;

public class MyPriorityQueue {

    private final LinkedList<Person>[] store;

    public MyPriorityQueue() {
        store = new LinkedList[3];
        store[Priority.LOW.ordinal()] = new LinkedList<>();
        store[Priority.NORMAL.ordinal()] = new LinkedList<>();
        store[Priority.HIGH.ordinal()] = new LinkedList<>();
    }

    public void add(Priority priority, Person person) {
        if (store[priority.ordinal()].contains(person)) {
            return;
        }
        store[priority.ordinal()].addLast(person);
    }

    public Person get() {
        if (!store[Priority.HIGH.ordinal()].isEmpty()) {
```

```
            return store[Priority.HIGH.ordinal()].removeFirst();
        }
        if (!store[Priority.NORMAL.ordinal()].isEmpty()) {
            return store[Priority.NORMAL.ordinal()].removeFirst();
        }
        if (!store[Priority.LOW.ordinal()].isEmpty()) {
            return store[Priority.LOW.ordinal()].removeFirst();
        }
        return null;
    }

    public static void main(String[] args) {
        Person p1 = new Person("A1", "Jim", "Black", LocalDate.of(1961,
Month.MARCH, 1));
        Person p2 = new Person("A2", "Jack", "Blue", LocalDate.of(1967,
Month.MARCH, 1));
        Person p3 = new Person("A3", "Jim", "Green", LocalDate.of(1971,
Month.MARCH, 1));
        Person p4 = new Person("A4", "Mary", "White", LocalDate.of(1961,
Month.MARCH, 1));
        Person p5 = new Person("A5", "Ann", "Red", LocalDate.of(1967, Month.MARCH,
1));
        Person p6 = new Person("A6", "Kelly", "Pink", LocalDate.of(1971,
Month.MARCH, 1));
        MyPriorityQueue pQueue = new MyPriorityQueue();
        pQueue.add(Priority.LOW, p1);
        pQueue.add(Priority.LOW, p2);
        pQueue.add(Priority.NORMAL, p3);
        pQueue.add(Priority.NORMAL, p4);
        pQueue.add(Priority.HIGH, p5);
        pQueue.add(Priority.HIGH, p6);
        Person p;
        while ((p = pQueue.get()) != null) {
            System.out.println(p);
        }
    }

}
```

Κώδικας 12.21 *Η κλάση MyPriorityQueue*

Όπως φαίνεται στον κώδικα 12.21, η κύρια δομή της κλάσης MyPriorityQueue είναι ένας πίνακας από LinkedList. Η κάθε λίστα του πίνακα φιλοξενεί αντικείμενα τύπου Person. Η Priority είναι απαριθμήσιμος τύπος με τρείς τιμές, LOW, NORMAL και HIGH.

Στον δημιουργό της MyPriorityQueue, δημιουργείται καταρχάς ο πίνακας store με σταθερό μέγεθος 3, δηλαδή ένα στοιχείο για κάθε δυνατή προτεραιότητα. Για κάθε στοιχείο του πίνακα δημιουργείται μια LinkedList την οποία χειριζόμαστε σαν δομή FIFO.

Η add λαμβάνει δύο παραμέτρους, την προτεραιότητα και τον ασθενή. Από την προτεραιότητα εντοπίζει την επιμέρους ουρά προτεραιότητας στο τέλος της οποίας προσθέτει τον ασθενή εφόσον δεν υπάρχει ήδη σε αυτήν την ουρά.

Η get ελέγχει τις επιμέρους ουρές προτεραιότητας. Αν βρει ουρά υψηλής προτεραιότητας, επιστρέφει τον ασθενή στην αρχή της ουράς και ταυτοχρόνως τον διαγράφει από την ουρά. Αν η ουρά υψηλής προτεραιότητας είναι άδεια, προχωράει και ελέγχει κατά σειρά την ουρά μεσαίας προτεραιότητας και την ουρά χαμηλής προτεραιότητας. Αν όλες οι ουρές είναι άδειες, επιστρέφει null. Επομένως, ένας τρόπος για να ελέγξουμε αν η MyPriorityQueue είναι άδεια είναι αν η get μας επιστρέψει null. Αυτόν ακριβως τον τρόπο αξιοποιούμε στην main καθώς αδειάζουμε το αντικείμενο pQueue.

12.8 Ασκήσεις προς Λύση

12.8.1 Διαχείριση χωρητικότητας στην κλάση MyHashMap

Στην κλάση MyHashMap έχουμε ορίσει τον πίνακα store να έχει σταθερό μέγεθος. Η επιλογή αυτή δεν είναι ιδανική. Ενδεχομένως ένα μέγεθος της τάξης των δεκάδων θέσεων να είναι ικανοποιητικό όσο οι εγγραφές που καταχωρούνται είναι σχετικά λίγες. Καθώς όμως αυξάνεται ο αριθμός των εγγραφών, όλο και περισσότερες εγγραφές αντιστοιχίζονται στην ίδια θέση του πίνακα, δηλαδή όλο και περισσότερα αντικείμενα Person καταχωρούνται στην ίδια λίστα. Προφανές αποτέλεσμα είναι η καθυστέρηση των λειτουργιών της MyHashMap. Η HashMap του πακέτου java.util λύνει αυτό το πρόβλημα με τη βοήθεια της εσωτερικής χωρητικότητας (capacity) και του συντελεστή φόρτωσης (load factor). Και τα δύο μεγέθη μπορεί να καθοριστούν κατά τη δημιουργία των αντικειμένων HashMap και ισχύει ο εξής κανόνας: Όταν ο αριθμός των εγγραφών στη δομή hash είναι μεγαλύτερος από το γινόμενο του συντελεστή φόρτωσης επί την εσωτερική χωρητικότητα, τότε η εσωτερική χωρητικότητα αυξάνεται αυτόματα έτσι ώστε να είναι περίπου διπλάσια από τον αριθμό των εγγραφών.

Προσθέστε ανάλογη συμπεριφορά στην κλάση MyHashMap.

12.8.2 Τραπεζική εφαρμογή

Η τράπεζα World Bank διατηρεί λογαριασμούς πελατών δύο τύπων, τους προθεσμιακούς και του ταμιευτηρίου. Κάθε λογαριασμός, ανεξαρτήτως τύπου, διαθέτει έναν αριθμό λογαριασμού και μπορεί να έχει έναν ή περισσότερους δικαιούχους. Επιπλέον, ένας πελάτης της τράπεζας μπορεί να είναι δικαιούχος σε έναν ή περισσότερους λογαριασμούς. Κάθε φορά που ένας πελάτης επιθυμεί να κάνει κατάθεση ή ανάληψη χρημάτων θα πρέπει να προσδιορίζει τον αριθμό λογαριασμού του που θα κινηθεί. Η τράπεζα ελέγχει και αν βρεθεί λογαριασμός με αυτόν τον αριθμό που έχει ως δικαιούχο τον συγκεκριμένο πελάτη, προχωράει στην κίνηση του λογαριασμού. Αν όμως δεν βρεθεί τέτοιος λογαριασμός, ρωτάει τον πελάτη αν επιθυμεί να του ανοίξει λογαριασμό. Σε περίπτωση αρνητικής απάντησης, το αίτημα του πελάτη ακυρώνεται. Αν όμως ο πελάτης συναινέσει στη δημιουργία λογαριασμού, τότε η τράπεζα ρωτά αν ο λογαριασμός θα είναι προθεσμιακός ή ταμιευτηρίου και προχωρά στη δημιουργία και κίνηση του λογαριασμού. Ο αριθμός λογαριασμού δημιουργείται από την τράπεζα και είναι μοναδικός για κάθε λογαριασμό της ίδιας τράπεζας.

Οι λογαριασμοί κινούνται χρεούμενοι ή πιστούμενοι, δηλαδή η κατάθεση χρημάτων προκαλεί πίστωση του λογαριασμού και η ανάληψη χρέωση. Όλες οι κινήσεις ενός λογαριασμού διατηρούνται αναλυτικά από την τράπεζα. Κάθε κίνηση πέραν του τύπου της, δηλαδή αν είναι χρεωστική ή πιστωτική, διατηρεί και την ημερομηνία κατά την οποία πραγματοποιήθηκε καθώς και μία περιγραφή. Το υπόλοιπο ενός λογαριασμού είναι το σύνολο των πιστώσεών του μείον το σύνολο των χρεώσεών του. Οι δε δικαιούχοι δεν δύνανται να αποσύρουν από λογαριασμό ποσό μεγαλύτερο από το υπόλοιπό του.

Η τράπεζα έχει τη δυνατότητα να παράγει αναφορά για όλους τους λογαριασμούς ή αν ζητηθεί από πελάτη να παράγει αναφορά ειδικά για τους λογαριασμούς του συγκεκριμένου πελάτη. Για κάθε λογαριασμό που περιλαμβάνεται στην αναφορά της τράπεζας εμφανίζονται ο αριθμός λογαριασμού, ο τύπος του, οι δικαιούχοι του, οι κινήσεις του και το υπόλοιπό του σύμφωνα με το υπόδειγμα που ακολουθεί.

```
2 Προθεσμιακός, Δικαιούχοι:  John Black Mary Smith
2019-10-20, Πίστωση, 1000.0, Καταθεση
2020-08-17, Πίστωση, 1000.0, Καταθεση
2020-02-24, Χρέωση, 500.0, Ανάληψη
2019-11-29, Χρέωση, 300.0, Ανάληψη
Υπόλοιπο = 1200.0
```

Να δημιουργηθεί εφαρμογή που

1. Δημιουργεί μία τράπεζα
2. Δημιουργεί μια σειρά από πελάτες
3. Δημιουργεί μια σειρά από λογαριασμούς. Κάποιοι θα πρέπει να είναι προθεσμιακοί και κάποιοι ταμιευτηρίου. Κάποιοι θα πρέπει να έχουν έναν δικαιούχο και κάποιοι περισσότερους από έναν.
4. Οι πελάτες να προβούν σε καταθέσεις και αναλήψεις. Οι κινήσεις των λογαριασμών να γίνουν με τυχαία ημερομηνία που να περιλαμβάνεται στις τελευταίες 1000 ημέρες πριν την ημερομηνία του συστήματος.

5. Η τράπεζα να εκδώσει αναφορά για το σύνολο των λογαριασμών της.
6. Η τράπεζα να εκδώσει αναφορά κατόπιν αιτήματος συγκεκριμένου πελάτη.

Για την αναπαράσταση των ποσών να χρησιμοποιηθεί η κλάση BigDecimal.

Υποδείξεις

- Καταρχάς προσπαθήστε να εντοπίσετε τις κλάσεις αυτής της εφαρμογής. Συνήθως οι κλάσεις αντιστοιχούν σε ουσιαστικά μέσα στο κείμενο των απαιτήσεων, στην περίπτωσή μας στην εκφώνηση της άσκησης.
- Προσπαθήστε να βρείτε τις ενέργειες στις οποίες προβαίνει η κάθε κλάση και κωδικοποιήστε τες ως συναρτήσεις. Συνήθως, οι ενέργειες αντιστοιχούν σε ρήματα στο κείμενο των απαιτήσεων.
- Φροντίστε ώστε η διεπαφή χρήστη να υποστηρίζεται αποκλειστικά από ειδική κλάση εφοδιασμένη με τις κατάλληλες στατικές μεθόδους.

12.8.3 Δημιουργία μοναδικών στιγμιότυπων

Η κλάση Alpha διαθέτει δύο μέλη-δεδομένα τύπου String, το id και το description. Κάθε αντικείμενο Alpha προσδιορίζεται μονοσήμαντα από το id του. Η εφαρμογή στην οποία αξιοποιείται η Alpha απαιτεί να μην είναι δυνατή η δημιουργία δύο στιγμιότυπων με το ίδιο id ακόμη και αν αναφέρονται στο ίδιο αντικείμενο. Να υλοποιηθεί η κλάση Alpha.

Υπόδειξη

Ορίστε ένα στατικό HashMap<String, Alpha> στην κλάση Alpha. Κάντε τον δημιουργό της ιδιωτικό. Προσθέστε μία συνάρτηση public Alpha create(String id, String description) που ελέγχει αν αντικείμενο Alpha με αυτό το id υπάρχει ήδη στο HashMap. Αν το βρει το επιστρέφει, αν όχι το δημιουργεί, το προσθέτει στο HashMap και το επιστρέφει.

Βιβλιογραφία

[1] D. Liang, Εισαγωγή στον Προγραμματισμό Java, 10η. Θεσσαλονίκη: Τζιόλας.

[2] E. Freeman, E. Robson, B. Bates, and K. Sierra, Head First Design Patterns, 1st edition. Sebastopol, CA: O'Reilly Media, 2004.

Κεφάλαιο 13

Σύνοψη
Στην ενότητα αυτήν παρουσιάζεται αναλυτικά η κληρονομικότητα, δηλαδή το δεύτερο από τα θεμελιώδη χαρακτηριστικά του αντικειμενοστρεφούς μοντέλου ενώ γίνεται εισαγωγή και στο τρίτο θεμελιώδες χαρακτηριστικό, τον πολυμορφισμό. Εξηγείται η δυνατότητα επανορισμού των κληρονομούμενων μεθόδων, ο ρόλος των λέξεων κλειδιών final και protected στο πλαίσιο της κληρονομικότητας, παρουσιάζεται η γονική κλάση Object, ο τελεστής instanceof και οι αφηρημένες κλάσεις. Συζητούνται κριτήρια επιλογής μεταξύ κληρονομικότητας και σύνθεσης.

Προαπαιτούμενη γνώση
Η ενσωμάτωση στο αντικειμενοστρεφές μοντέλο με Java και οι βασικές προκαθορισμένες κλάσεις, δηλαδή οι String, StringBuilder, ArrayList, LinkedList, Stack, PriorityQueue, HashSet, HashMap, HashTree, LocalDate, Localtime, Period και Duration.

Λέξεις κλειδιά
Κληρονομικότητα (Inheritance), Πολυμορφισμός (Polymorphism), Γονική κλάση (parent class), Παράγωγη κλάση (derived class), Αφηρημένη κλάση (abstract class).

13 Κληρονομικότητα και Πολυμορφισμός

Όπως έχουμε αναφέρει στην ενότητα 10, η κληρονομικότητα μας παρέχει τη δυνατότητα να συσχετίζουμε τις κλάσεις με σχέσεις ιεραρχίας και πιο συγκεκριμένα με σχέσεις προγόνου-απογόνου. Επάνω στην κληρονομικότητα βασίζεται το άλλο πανίσχυρο χαρακτηριστικό του αντικειμενοστρεφούς μοντέλου, ο πολυμορφισμός. Με τον πολυμορφισμό μπορούμε να διαχειριζόμαστε αντικείμενα μιας κλάσης απογόνου με αναφορές τύπου κάποιας προγονικής κλάσης ενώ ταυτόχρονα απολαμβάνουμε της προστασίας του συστήματος ελέγχου των τύπων. Στη συνέχεια παρουσιάζονται τα δύο αυτά χαρακτηριστικά και αναλύονται οι ιδιότητές τους και τα πλεονεκτήματά τους.

13.1 Η Κληρονομικότητα

Έστω η κλάση Root του κώδικα 13.1.

```
public class Root {
    private int id;
    private String Description;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getDescription() {
        return Description;
    }

    public void setDescription(String Description) {
        this.Description = Description;
    }

    @Override
    public int hashCode() {
        int hash = 7;
```

```
        hash = 89 * hash + this.id;
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Root other = (Root) obj;
        return this.id==other.id;
    }

    @Override
    public String toString() {
        return id + ", " + Description;
    }
}
```

Κώδικας 13.1 *Η κλάση Root*

```
public class LeafA extends Root {
    public static void main(String[] args) {
        LeafA cA=new LeafA();
        System.out.println("getters: "+cA.getId()+" "+cA.getDescription());
        cA.setDescription("LeafA Object");
        System.out.println("toString: "+cA);
    }
}
```

Κώδικας 13.2 *Η κλάση LeafA κληρονομεί όλα τα χαρακτηριστικά της Root*

Στον κώδικα 13.2 καταρχάς δηλώνουμε ένα αντικείμενο της LeafA. Στη συνέχεια καλούμε την getId και την getDescription. Ωστόσο τέτοιες συναρτήσεις δεν έχουμε ορίσει στην LeafA. Παρόλα αυτά, ο κώδικας τρέχει καθώς η LeafA έχει στη διάθεσή της τις getId και getDescription που έχει κληρονομήσει από την Root. Το ίδιο ισχύει και για την setDescription αλλά και για την toString που καλείται αυτομάτως στην τελευταία γραμμή. Η έξοδος του κώδικα 13.2 έχει ως εξής:

```
getters: 0 null
toString: 0, LeafA Object
```

Αρχικά, η getId επιστρέφει 0 και η getDescription επιστρέφει null. Η LeafA εκτός από τις συναρτήσεις κληρονομεί και τις μη στατικές μεταβλητές της Root, το id και το description. Οι μεταβλητές αυτές δεν έχουν αρχικοποιηθεί στην Root και έτσι έχουν τις εξ' ορισμού τιμές ως μεταβλητές στιγμιότυπου. Στη συνέχεια, με την κλήση της setDescription δίνουμε την τιμή LeafA Object στην description που παρουσιάζει η toString της τελευταίας γραμμής.

Η κλάση Root ονομάζεται γονική κλάση (parent class) ή βασική κλάση (Base class) γιατί από αυτήν κληρονομεί τα χαρακτηριστικά της η LeafA. Η δε LeafA ονομάζεται και κλάση-παιδί (child class) ή παράγωγη κλάση (derived class).

Αν προσθέσουμε τον ακόλουθο δημιουργό στην Root

```
public Root() {
     System.out.println("Root constructor");
```

```
        this.id = 1;
        this.Description = "Parent";
    }
```

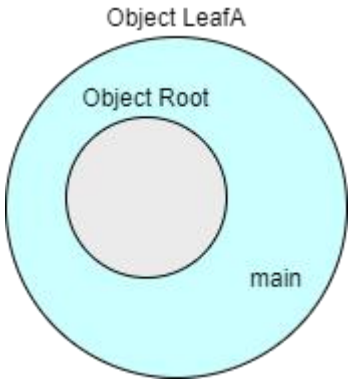
τότε η έξοδος του κώδικα 13.2 θα μεταβληθεί ως εξής:

```
Root constructor
getters: 1 Parent
toString: 1, ChildA Object
```

Βλέπουμε δηλαδή πώς κλήθηκε ο δημιουργός της κλάσης Root, παρότι δεν δημιουργήσαμε σαφώς ένα αντικείμενο Root, και έδωσε τις τιμές 1 και Parent στις μεταβλητές id και description, αντίστοιχα.
Αν προσθέσουμε και έναν δημιουργό με παραμέτρους στην Root, π.χ.

```
public Root(int id, String Description) {
        this.id = id;
        this.Description = Description;
}
```

μπορούμε εύκολα να διαπιστώσουμε ότι η έξοδος του κώδικα 13.2 δεν θα μεταβληθεί. Αν όμως διαγράψουμε τον δημιουργό χωρίς παραμέτρους της Root και αφήσουμε μόνο τον δημιουργό με παραμέτρους θα διαπιστώσουμε πως η LeafA δεν μεταγλωττίζεται.
Πώς εξηγείται αυτή η συμπεριφορά; Κάθε αντικείμενο της παράγωγης κλάσης LeafA περικλείει ένα αντικείμενο την γονικής κλάσης Root όπως δείχνει το σχήμα 13.1.



Σχήμα 13.1 *Η παράγωγη κλάση περικλείει ένα αντικείμενο της γονικής.*

Πέραν των συναρτήσεων και μεταβλητών που κληρονομεί η παράγωγος κλάση από τη γονική, μπορεί να διαθέτει επιπλέον συναρτήσεις ή και μεταβλητές. Συγκεκριμένα, η κλάση LeafA διαθέτει μια μέθοδο main. Όπως φαίνεται στο σχήμα 13.1, τα αντικείμενα της παράγωγης περιέχουν ένα αντικείμενο της γονικής και τα μέλη, δηλαδή την main στο παράδειγμά μας, που ανήκουν αποκλειστικά στην παράγωγη. Κατά την κατασκευή των αντικειμένων LeafA, ο δημιουργός της κλάσης καλεί τον δημιουργό της γονικής προκειμένου να κατασκευάσει το εσωτερικό αντικείμενο Root. Όταν η γονική διαθέτει δημιουργό χωρίς παραμέτρους, τότε η κλήση του μπορεί να αυτοματοποιηθεί. Όταν όμως η γονική κλάση δεν διαθέτει δημιουργό χωρίς παραμέτρους, τότε η κλήση του δημιουργού θα πρέπει να πραγματοποιηθεί σαφώς, καθώς απαιτείται να δοθούν τιμές στις παραμέτρους του. Η κλήση αυτή γίνεται στον δημιουργό της παράγωγης με τη βοήθεια της λέξης-κλειδί super, ως εξής:

```
public LeafA(int id, String Description) {
        super(id, Description);
}
```

Επομένως σε αυτήν την περίπτωση, η παράγωγη κλάση υποχρεωτικά πρέπει να διαθέτει έναν δημιουργό ο οποίος να είναι σε θέση να καλέσει τον δημιουργό της γονικής και να δώσει τιμές στις παραμέτρους του. Στην

τυπική περίπτωση, οι παράμετροι του δημιουργού της γονικής είναι και παράμετροι του δημιουργού της παράγωγης. Αν ωστόσο η παράγωγη έχει και άλλα μέλη που πρέπει να αρχικοποιηθούν στον δημιουργό, τότε αυτό θα γίνει μετά την κλήση του δημιουργού της γονικής. Με άλλα λόγια, η κλήση του δημιουργού της γονικής απαιτείται να είναι η πρώτη εντολή στον δημιουργό της παράγωγης. Εκείνο που πρέπει να τονιστεί είναι πως οι δημιουργοί δεν κληρονομούνται, δηλαδή δεν μπορούμε να καλέσουμε τον δημιουργό γονικής για να κατασκευάσουμε αντικείμενο παράγωγης. Αυτό είναι αναγκαία συνέπεια ώστε η κλήση ενός δημιουργού να κατασκευάζει με σαφήνεια αντικείμενο συγκεκριμένης κλάσης.
Με τη βοήθεια του νέου δημιουργού της, η παράγωγη μπορεί να δημιουργήσει αντικείμενα όπως επιδεικνύει ο κώδικας 13.3:

```
public static void main(String[] args) {
    LeafA cA=new LeafA(4,"LeafA");
    System.out.println("getters: "+cA.getId()+" "+cA.getDescription());
    cA.setDescription("ChildA Object");
    System.out.println("toString: "+cA);
}
```

Κώδικας 13.3 *Δημιουργία παράγωγων αντικειμένων όταν η γονική διαθέτει μόνο δημιουργούς με παραμέτρους*

Η έξοδος του κώδικα 13.3 έχει ως εξής:

```
getters: 4 LeafA
toString: 4, ChildA Object
```

Παρότι η LeafA κληρονομεί όλες τις μεταβλητές και συναρτήσεις της Root, η πρόσβαση στα ιδιωτικά μέλη της Root δεν είναι δυνατή από την LeafA, δηλαδή ο κώδικας

```
System.out.println(cA.id);
```

προκαλεί λάθος μεταγλώττισης.
Ο κώδικας 13.4 επιδεικνύει μια άλλη κλάση, επίσης παράγωγη της Root.

```
public class LeafB extends Root {
    private double value;

    public LeafB(double value, int id, String Description) {
        super(id, Description);
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    public void setValue(double value) {
        this.value = value;
    }

    public static void main(String[] args) {
        LeafB leafB=new LeafB(120,5,"LeafB");
        System.out.println(leafB.getId()+" "+leafB.getDescription()+"
"+leafB.getValue());
    }
}
```

Κώδικας 13.4 *Παράγωγη κλάση με επιπλέον χαρακτηριστικά*

Η κλάση LeafB κληρονομεί από την κλάση Root, διαθέτει όμως μια επιπλέον μεταβλητή στιγμιότυπου, την value. Για τη διαχείριση της μεταβλητής, η κλάση ορίζει τους κατάλληλους ρυθμιστές και αναγνώστες. Ο

δημιουργός της κλάσης καλεί τον δημιουργό της γονικής για να κατασκευάσει το εσωτερικό αντικείμενο και στη συνέχεια δίνει τιμή στην επιπλέον μεταβλητή value.
Η έξοδος της main του κώδικα 13.4 έχει ως εξής:

```
5 LeafB 120.0
```

Η κληρονομικότητα δεν περιορίζεται σε ένα επίπεδο. Αντίθετα, δεν υπάρχει κανένας περιορισμός ούτε κατά βάθος ούτε κατά πλάτος. Για παράδειγμα, η κλάση των ηλεκτρικών συσκευών, μπορεί να έχει πολλούς απογόνους στο ίδιο επίπεδο, π.χ. υπολογιστές, πλυντήρια, τηλεοράσεις, ραδιόφωνα, αλλά και πολλούς απογόνους κάθετα, π.χ. ο υπολογιστής μπορεί να έχει ως απογόνους τον επιτραπέζιο υπολογιστή και τον φορητό υπολογιστή, ενώ ο φορητός υπολογιστής μπορεί να έχει απογόνους τον υπολογιστή χειρός (calculator) και την ταμπλέτα (tablet). Στην πράξη, συνήθως διαμορφώνονται ιεραρχίες κλάσεων που συνδέονται μεταξύ τους με κληρονομικότητα. Οι ιεραρχίες κληρονομικότητας συναντώνται εκτεταμένα και στις βιβλιοθήκες της Java.

13.1.1 Επανορισμός κληρονομούμενων μεθόδων

Ποια θα είναι όμως η έξοδος αν μετατρέψουμε την main της LeafB όπως δείχνει ο κώδικας 13.5;

```
public static void main(String[] args) {
    LeafB leafB=new LeafB(120,5,"LeafB");
    System.out.println(leafB);
}
```

Κώδικας 13.5 *Η main καλεί την toString της γονικής*

Εδώ εμφανίζουμε κατευθείαν ένα αντικείμενο LeafB. Όπως γνωρίζουμε, σε αυτές τις περιπτώσεις καλείται η μέθοδος toString για να επιστρέψει την αναπαράσταση του αντικειμένου με τη μορφή συμβολοσειράς. Ωστόσο, στην LeafB δεν έχουμε ορίσει μέθοδο toString. Επομένως αυτή κληρονομείται από την Root. Η toString της Root βεβαίως δεν συμπεριλαμβάνει τη μεταβλητή value. Συνεπώς, η έξοδος της main είναι:

```
5, LeafB
```

Αν μια μέθοδος της γονικής δεν καλύπτει τις ανάγκες της παράγωγης, τότε η μέθοδος αυτή μπορεί να επανοριστεί (override) στο επίπεδο της παράγωγης. Αν, επομένως, η toString της Root δεν μας καλύπτει για την LeafB, τότε θα την επανορίσουμε στην LeafB. Κατά τον επανορισμό της μεθόδου, μπορούμε εφόσον μας διευκολύνει να χρησιμοποιήσουμε τη μέθοδο της γονικής και να προσθέσουμε την επιπλέον απαιτούμενη λειτουργικότητα, όπως δείχνει ο κώδικας 13.6.

```
@Override
public String toString() { //Κώδικας 13.6
    return super.toString() + ", " + value;
}
```

Κώδικας 13.6 *Η toString επανοριζόμενη στην παράγωγη κλάση LeafB*

Προκειμένου να καλέσουμε τη μέθοδο της γονικής από την παράγωγη, χρησιμοποιούμε και πάλι τη λέξη κλειδί super αλλά με διαφορετικό συντακτικό από ότι στον δημιουργό. Ο κώδικας 13.5 τώρα έχει την ακόλουθη έξοδο:

```
5, LeafB, 120.0
```

Η σειρά @Override είναι ένα σχόλιο (annotation) προς τον μεταγλωττιστή με το οποίο πληροφορείται ότι η μέθοδος που ακολουθεί είναι επανορισμός κληρονομούμενης μεθόδου. Το σχόλιο αυτό δεν είναι υποχρεωτικό, είναι όμως χρήσιμο. Για παράδειγμα, αν επιχειρώντας να επανορίσουμε τη μέθοδο toString, δηλώσουμε κατά λάθος μέθοδο asString(), τότε προσθέτουμε μια νέα μέθοδο και δεν επανορίζουμε την toString. Ως αποτέλεσμα, τα αντικείμενα LeafB θα συνεχίσουν να εμφανίζονται χωρίς την τιμή της value.

Ένα άλλο παράδειγμα. Αν ορίσουμε μια μέθοδο String toString(double value), τότε έχουμε υπερφορτώσει τη μέθοδο toString, δεν την έχουμε όμως επανορίσει. Με την @Override, ο μεταγλωττιστής μπορεί να ανιχνεύσει τέτοιου είδους σφάλματα και να μας προστατεύσει από αυτά.

Προσέξτε πως δεν τίθεται θέμα επανορισμού στην παράγωγη των ιδιωτικών μεθόδων της γονικής, καθώς για αυτές δεν επιτρέπεται η πρόσβαση στην παράγωγη κλάση. Επομένως, αν ορίσουμε μια μέθοδο στην παράγωγη με ακριβώς την ίδια διεπαφή με ιδιωτική μέθοδο της γονικής, έχουμε ορίσει απλώς μια νέα μέθοδο. Επιπλέον, κατά τον επανορισμό μιας μεθόδου, έχουμε τη δυνατότητα να ορίσουμε ευρύτερο προσδιοριστή προσπέλασης. Για παράδειγμα, μια μέθοδος που στη γονική έχει πρόσβαση πακέτου, μπορεί στην παράγωγη να έχει δημόσια πρόσβαση.

Οι στατικές μέθοδοι και μεταβλητές δεν συνδέονται με τα αντικείμενα αλλά με την ίδια την κλάση. Έτσι, τα στατικά μέλη δεν κληρονομούνται ούτε και επανορίζονται. Αν οριστεί σε παράγωγη κλάση στατική μέθοδος με ίδια ταυτότητα με στατική μέθοδο της γονικής, τότε πρόκειται για δύο διαφορετικές μεθόδους. Η διάκρισή τους από τον μεταγλωττιστή γίνεται με βάση το όνομα της κλάσης.

13.1.2 Final και protected

Αν επιθυμούμε να μην επιτρέπουμε στις παράγωγες κλάσεις να έχουν τη δυνατότητα να επανορίσουν μια μέθοδο της γονικής, τότε μπορούμε να χαρακτηρίσουμε αυτήν τη μέθοδο ως final, όπως δείχνει το παρακάτω απόσπασμα κώδικα:

```
public final int getId() {
     return id;
}
```

Αν πάλι επιθυμούμε να αποτρέψουμε τη δημιουργία παράγωγων κλάσεων, τότε μπορούμε να χαρακτηρίσουμε ως final ολόκληρη την κλάση, όπως δείχνει το απόσπασμα κώδικα που ακολουθεί:

```
public final class LeafA extends Root {}
```

Η κλάση LeafA είναι παράγωγη της κλάσης Root, αλλά δεν μπορεί να δημιουργηθεί κλάση παράγωγη της LeafA.
Είπαμε στην ενότητα 13.1 πως τα μέλη id και description είναι ιδιωτικά στην κλάση Root και επομένως δεν επιτρέπεται η απευθείας πρόσβαση σε αυτά από την παράγωγη LeafA. Αν επιθυμούμε τα μέλη αυτά να προστατεύονται από τρίτους, ωστόσο, οι παράγωγες κλάσεις να έχουν πρόσβαση και σε αυτά, τότε μπορούμε να τα χαρακτηρίσουμε ως protected, όπως δείχνει το ακόλουθο απόσπασμα κώδικα:

```
protected int id;
protected String Description;
```

Χρησιμοποιούμε δηλαδή τον προσδιοριστή προσπέλασης protected στη θέση του private. Ας σημειωθεί πως protected μπορεί να χαρακτηρισθεί και μια κλάση, εμφωλευμένη ή μη.

13.1.3 Κληρονομικότητα ή σύνθεση;

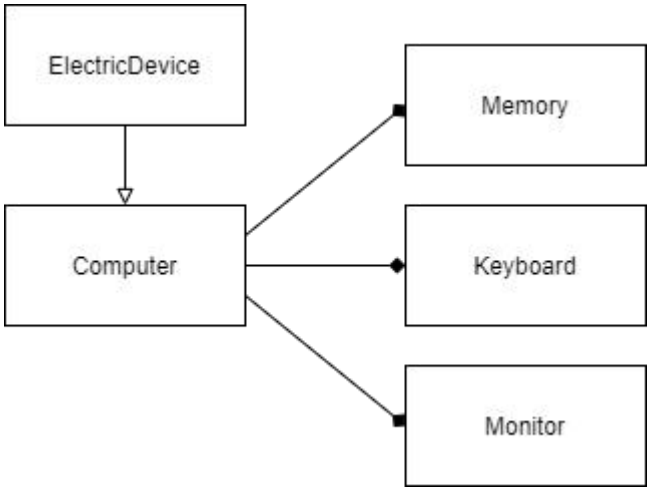
Δύο είναι οι βασικές σχέσεις μεταξύ των κλάσεων που θα μας απασχολήσουν στην ενότητα αυτήν, η σχέση “είναι ένα” (is a) και η σχέση “έχει ένα” (has a). Ένα ραδιόφωνο είναι μία ηλεκτρική συσκευή. Επομένως, ένα ραδιόφωνο διαθέτει όλα τα χαρακτηριστικά και λειτουργίες μιας ηλεκτρικής συσκευής, π.χ. μια ηλεκτρική συσκευή μπορεί να είναι συνδεδεμένη στο ρεύμα ή όχι, το ίδιο ισχύει και για ένα ραδιόφωνο. Ένα αυτοκίνητο είναι ένα όχημα. Ένα όχημα μπορεί να κινηθεί, το ίδιο και ένα αυτοκίνητο. Είναι λοιπόν σαφές πως η ηλεκτρική συσκευή και το ραδιόφωνο συνδέονται με σχέση κληρονομικότητας, δηλαδή το ραδιόφωνο ως κλάση μπορεί να κληρονομεί χαρακτηριστικά και λειτουργίες της ηλεκτρικής συσκευής. Το ίδιο ισχύει και μεταξύ των κλάσεων που αναπαριστούν το όχημα και το αυτοκίνητο.

Από την άλλη πλευρά, ένα αυτοκίνητο έχει έναν κινητήρα. Επίσης, έχει μηχανισμό ταχυτήτων. Η σχέση “έχει ένα” υποδηλώνει σύνθεση (composition) και όχι κληρονομικότητα, δηλαδή η κλάση αυτοκίνητο

μπορεί να περιλαμβάνει ένα μέλος της κλάσης που αναπαριστά τους κινητήρες και ένα μέλος της κλάσης που αναπαριστά τους μηχανισμούς ταχυτήτων.

Ας το δούμε με ένα παράδειγμα. Έστω η κλάση Computer αποτελεί μια απλή προσομοίωση ενός υπολογιστή. Αυτού του τύπου οι υπολογιστές υποστηρίζουν μόνο τρεις εντολές, την keyboardToMem που διαβάζει από το πληκτρολόγιο ένα String και το μεταφέρει στη μνήμη, την MemToMonitor, που εμφανίζει στην οθόνη τα περιεχόμενα της μνήμης και την clearMem που αρχικοποιεί τη μνήμη. Επομένως, ο υπολογιστής πρέπει να έχει μνήμη, πληκτρολόγιο και οθόνη. Ταυτόχρονα όμως ο υπολογιστής μας είναι και μια ηλεκτρική συσκευή και όπως όλες οι ηλεκτρικές συσκευές μπορεί να είναι συνδεδεμένος στο ρεύμα ή όχι.

Από την περιγραφή του παραδείγματός μας είναι φανερό πως ο υπολογιστής συνδέεται με τις κλάσεις που αναπαριστούν τη μνήμη, την οθόνη και το πληκτρολόγιο με τη σχέση “έχει ένα”, ενώ με τις ηλεκτρικές συσκευές με τη σχέση “είναι ένα”. Το παρακάτω διάγραμμα σε UML (Unified Modelling Language) [1] δείχνει τις σχέσεις μεταξύ των κλάσεων.



Σχήμα 13.2 *Σχέσεις σύνθεσης και κληρονομικότητας*

Στο σχήμα 13.2 το βελάκι με τριγωνική απόληξη χωρίς γέμισμα συμβολίζει κληρονομικότητα, δηλαδή ο Computer κληρονομεί από την ElectricDevice. Αντίθετα, το βελάκι με συμπαγή απόληξη σε σχήμα ρόμβου συμβολίζει σύνθεση, δηλαδή ο Computer έχει Memory, Keyboard και Monitor.

Ας αναπτύξουμε τώρα τις κλάσεις. Καταρχάς, χρειαζόμαστε την κλάση ElectricDevice. Η κλάση αυτή διαθέτει μία κύρια λειτουργία η οποία τη συνδέει ή την αποσυνδέει από το ρεύμα και μια λειτουργία που ελέγχει αν είναι ή όχι συνδεδεμένη στο ρεύμα, όπως δείχνει ο κώδικας 13.7.

```
public class ElectricDevice  {

    private boolean powerOn;

    public boolean isPowerOn() {
        return powerOn;
    }

    public void setPowerOn(boolean powerOn) {
        this.powerOn = powerOn;
    }
}
```

Κώδικας 13.7 *Η κλάση ElectricDevice*

Ας δούμε τώρα τις κλάσεις Keyboard, Monitor και Memory.

```
public class Keyboard {
```

```
        public String read() {
              return "ABCD";
        }
}
```

Η κλάση Keyboard διαθέτει μια απλή μέθοδο read που για απλότητα επιστρέφει μια συμβολοσειρά. Μπορείτε να αναπτύξετε την read ώστε πράγματι να λαμβάνει μια είσοδο από το πληκτρολόγιο και να την επιστρέφει. Όπως και να έχει, δεν επηρεάζονται οι έννοιες που παρουσιάζουμε εδώ.

```
public class Memory {

        private String content=null;

        public String getContent() {
              return content;
        }

        public void setContent(String content) {
              this.content = content;
        }

        public boolean isEmpty() {
              return content==null;
        }
}
```

Η κλάση Memory έχει τρεις λειτουργίες. Μπορεί να αποθηκεύσει μια τιμή τύπου String, να επιστρέψει την αποθηκευμένη τιμή και να ελέγξει αν υπάρχει ή όχι αποθηκευμένη τιμή.
Η κλάση Monitor μπορεί απλώς να εμφανίσει μια τιμή String.

```
public class Monitor {

        public void display(String data) {
              System.out.println(data);
        }
}
```

Τέλος, στον κώδικα 13.8 παρουσιάζεται η κλάση Computer:

```
public class Computer extends ElectricDevice {

    private final Keyboard keyboard = new Keyboard();
    private final Memory memory = new Memory();
    private final Monitor monitor = new Monitor();

    void keyboardToMem() {
        if (!isPowerOn()) {
            return;
        }
        String data = keyboard.read();
        memory.setContent(data);
    }

    void memToMonitor() {
        if (!isPowerOn() || memory.isEmpty()) {
            return;
        }
```

```
            monitor.display(memory.getContent());
        }

        void clearMem() {
            memory.setContent(null);
        }

        public void operate() {
            clearMem();
            keyboardToMem();
            memToMonitor();
        }

        public static void main(String[] args) {
            Computer alfaX = new Computer();
            alfaX.setPowerOn(true);
            alfaX.operate();
            alfaX.setPowerOn(false);
        }
    }
```

Κώδικας 13.8 *Η κλάση Computer*

Όπως φαίνεται στον κώδικα 13.8, η κλάση Computer κληρονομεί την κλάση ElectricDevice, ενώ διαθέτει ιδιωτικά μέλη των κλάσεων Memory, Keyboard και Monitor. Έτσι, ο Computer alfaX που δημιουργείται στην main, μπορεί να συνδεθεί και να αποσυνδεθεί από το ρεύμα με τη βοήθεια της setPowerOn που κληρονομεί από τις ηλεκτρικές συσκευές, να μεταφέρει την είσοδο του πληκτρολογίου του στη μνήμη του, να στέλνει το περιεχόμενο της μνήμης του στην οθόνη του, να αρχικοποιεί το περιεχόμενο της μνήμης του.

Πέραν των δύο βασικών σχέσεων, της σύνθεσης και κληρονομικότητας που μας ενδιαφέρουν στο πλαίσιο αυτού του βιβλίου, έχουν θεσπιστεί και άλλοι τύποι σχέσεων ώστε να υπάρχει η δυνατότητα οι σχέσεις μεταξύ των κλάσεων να προσδιορίζονται με μεγαλύτερη λεπτομέρεια [1].

13.1.4 Η κλάση Object

Η κλάση Object είναι η γονική κλάση κάθε κλάσης της Java. Τόσο οι κλάσεις της βιβλιοθήκης, όσο και οι οριζόμενες από τον χρήστη κλάσεις κληρονομούν αυτομάτως, χωρίς δηλαδή να το δηλώσουμε σαφώς με τη λέξη-κλειδί extends, την κλάση Object. Το ίδιο ισχύει και για τους πίνακες της Java οι οποίοι στην ουσία συνιστούν κλάσεις για τις οποίες προσφέρεται ένα διαφορετικό συντακτικό δημιουργίας και διαχείρισης.

Η κλάση Object περιλαμβάνει καταρχάς τις γνώριμές μας συναρτήσεις toString, equals και hashCode. Η συνάρτηση toString επιστρέφει μια συμβολοσειρά που αποτελείται από το όνομα του πακέτου ακολουθούμενο από το όνομα της κλάσης συνδεδεμένο με τον χαρακτήρα '@' και τον κωδικό κατακερματισμού του αντικειμένου εκφρασμένο σε μη προσημασμένη δεκαεξαδική μορφή. Αν τυπώσετε ένα αντικείμενο της κλάσης, θα λάβετε κάτι σαν java.lang.Object@4f3f5b2. Αν δημιουργήσετε μια κλάση, η συνάρτηση toString είναι αυτομάτως διαθέσιμη, καθώς κληρονομείται από την κλάση Object. Το ίδιο ισχύει για όλες τις συναρτήσεις της Object. Όποτε δεν μας διευκολύνει η toString, πράγμα που ισχύει συνήθως, μπορούμε να την επανορίσουμε στο επίπεδο της κλάσης που δημιουργήσαμε.

Η συνάρτηση equals της Object, για 2 μεταβλητές, o1 και o2 τύπου Object, θα επιστρέψει true μόνον εφόσον o1==o2, δηλαδή μόνον εφόσον οι δύο μεταβλητές αναφέρονται στην ίδια θέση μνήμης ή αλλιώς στο ίδιο αντικείμενο. Αυτή η σύγκριση τις περισσότερες φορές δεν μας εξυπηρετεί, καθώς αντικείμενα με το ίδιο περιεχόμενο μπορεί να έχουν κατανεμηθεί σε διαφορετικές θέσεις μνήμης. Έτσι, αποτελεί σχεδόν κανόνα ο επανορισμός της equals στις κλάσεις που δημιουργούμε. Εξαίρεση σε αυτόν τον κανόνα αποτελούν κλάσεις που εγγυώνται πως μόνο ένα αντικείμενο με συγκεκριμένο περιεχόμενο μπορεί να δημιουργηθεί κατά την εκτέλεση της εφαρμογής. Χαρακτηριστικό παράδειγμα είναι οι απαριθμήσιμοι τύποι οι οποίοι διατηρούν τον δημιουργό τους ιδιωτικό, αποτρέποντας έτσι τον χρήστη τους να δημιουργεί διαφορετικά στιγμιότυπα του ίδιου αντικειμένου. Ας θεωρήσουμε ένα απαριθμήσιμο τύπο για τις ημέρες της εβδομάδας με αντικείμενα SUNDAY έως SATURDAY. Υπάρχει εγγύηση πως μόνο ένα αντικείμενο, π.χ. SUNDAY έχει δημιουργηθεί. Μπορεί να έχουμε πολλές αναφορές σε αυτό το αντικείμενο. Το ίδιο το αντικείμενο όμως υπάρχει μια μόνο φορά στη μνήμη της εφαρμογής μας. Σε αυτήν την περίπτωση δεν υπάρχει ανάγκη επανορισμού της equals.

Μια διαφορετική περίπτωση είναι κλάσεις που χρησιμοποιούνται ως πρόχειρες μεταβλητές για διαμόρφωση δεδομένων, π.χ. η κλάση StringBuilder. Τα αντικείμενα StringBuilder χρησιμοποιούνται για τη διαμόρφωση αντικειμένων String. Μπορεί σε κάποια φάση κατά τη διαμόρφωσή τους δύο αντικείμενα StringBuilder να παρουσιάζουν ίδιο περιεχόμενο, αργότερα όμως να καταλήξουν σε διαφορετικά περιεχόμενα. Επομένως, θεωρήθηκε πως δεν έχει νόημα η σύγκριση του περιεχομένου δύο αντικειμένων StringBuilder. Εξάλλου, αν κάπου χρειαστείτε τέτοια σύγκριση, μπορείτε πάντα να καλέσετε sb1.toString().equals(sb2.toString()), όπου sb1 και sb2 αντικείμενα StringBuilder.

Άλλες συναρτήσεις της Object που θα πρέπει να αναφέρουμε εδώ είναι οι:

```
protected Object clone()
```

Επιστρέφει ένα αντίγραφο του αντικειμένου που την καλεί. Η συνάρτηση έχει προστατευμένη πρόσβαση και επομένως είναι διαθέσιμη σε κάθε κλάση της Java, αφού όλες οι κλάσεις έχουν ως γονική την Object. Ωστόσο, για να χρησιμοποιηθεί η συνάρτηση θα πρέπει η κλάση να υλοποιεί τη διεπαφή Cloneable, διαφορετικά θα παράγει εξαίρεση. Για τις διεπαφές θα μιλήσουμε αναλυτικά στην ενότητα 14. Η περίεργη αυτή συμπεριφορά οφείλεται στο γεγονός πως η συνάρτηση ορίστηκε στην κλάση Object και εν συνεχεία έγινε αντιληπτό πως υπάρχει η ανάγκη για κάποιες κλάσεις να αποτρέπεται η δημιουργία αντιγράφων των αντικειμένων τους.

```
public final Class<?> getClass()
```

Στην ουσία επιστρέφει την κλάση στην οποία ανήκει το αντικείμενο που την καλεί. Όπως φαίνεται από τη διεπαφή της, η τιμή επιστροφής της είναι μια γενικευμένη κλάση. Έχοντας ένα τέτοιο αντικείμενο μπορούμε να προσπελάσουμε χαρακτηριστικά της κλάσης που ανήκει αυτό το αντικείμενο, π.χ.

```
ElectricDevice eD = new ElectricDevice();
System.out.println(eD.getClass().getName());
```

13.1.5 Η σειρά αρχικοποίησης

Τα μέλη της κλάσης σε αντίθεση με τις τοπικές μεταβλητές αρχικοποιούνται αυτόματα. Επιπλέον, μπορεί ο ιδιοκτήτης της κλάσης να τα αρχικοποιήσει σαφώς εκεί που τα δηλώνει. Πέραν αυτού είναι δυνατόν σε μία κλάση να προστεθεί ένα ή περισσότερα μπλοκ αρχικοποίησης (Initialization block). Τα μπλοκ αρχικοποίησης δεν έχουν και ιδιαίτερη χρησιμότητα και θα τα συναντήσετε πολύ σπάνια σε κώδικες Java. Ωστόσο, αν χρησιμοποιηθούν θα πρέπει να διαφοροποιούνται αυτά για τις στατικές μεταβλητές από τα μπλοκ για τις μεταβλητές στιγμιότυπου. Ο κώδικας που ακολουθεί επιδεικνύει αυτές τις πιθανές αρχικοποιήσεις:

```
import java.util.ArrayList;

public class Initialization {

     ArrayList<String> store = new ArrayList<>();
     String id, otherData;
     static boolean check;

     static { //Initialization Block for static members
          check = true;
     }

     { //Initialization Block
          otherData = "OtherData";
     }

     public Initialization(String id) {
          this.id = id;
     }
```

```
    }
```

Πρώτα από όλα αρχικοποιείται το στατικό μέλος check. Το μέλος store αρχικοποιείται σαφώς κατά τη δήλωσή του. Στη συνέχεια τρέχει το μπλοκ αρχικοποίησης όπου αρχικοποιείται το μέλος otherData. Τέλος το id αρχικοποιείται στον δημιουργό της κλάσης. Τι ισχύει όμως όταν η κλάση είναι παράγωγη; Γενικά, η προτεραιότητα αρχικοποίησης έχει ως εξής:

1. Στατικά πεδία και στατικά μπλοκ προγόνων.
2. Στατικά πεδία και στατικό μπλοκ της τρέχουσας κλάσης.
3. Μεταβλητές στιγμιότυπου και μπλοκ αρχικοποίησης προγόνων.
4. Δημιουργός του προγόνου.
5. Μεταβλητές στιγμιότυπου και μπλοκ αρχικοποίησης της τρέχουσας κλάσης.
6. Δημιουργός της τρέχουσας τάξης.

Δύο ή περισσότερα στοιχεία με την ίδια προτεραιότητα αρχικοποιούνται σύμφωνα με τη σειρά τοποθέτησής τους στον κώδικα.

13.2 Πολυμορφισμός

Ένα προφανές πλεονέκτημα της κληρονομικότητας είναι η επαναχρησιμοποίηση κώδικα (code reusability). Οι μέθοδοι της γονικής κλάσης είναι χρήσιμες και στις παράγωγες. Με την κληρονομικότητα γίνονται αυτόματα διαθέσιμες και έτσι δεν χρειάζεται να δημιουργήσουμε αντίγραφα του ίδιου κώδικα.
Το μεγαλύτερο όμως πλεονέκτημα της κληρονομικότητας είναι πως υποστηρίζει τον πολυμορφισμό. Ο πολυμορφισμός βασίζεται στη δυνατότητα να εκχωρήσουμε σε μια μεταβλητή αναφοράς τύπου μιας γονικής κλάσης, ένα στιγμιότυπο παράγωγης. Για παράδειγμα, η εκχώρηση

```
ElectricDevice betaX = new Computer();
```

είναι απολύτως ορθή. Εφόσον ένας ηλεκτρονικός υπολογιστής είναι μια ηλεκτρική συσκευή, γιατί να μην μπορούμε να τον διαχειριστούμε σαν ηλεκτρική συσκευή; Από τη στιγμή όμως που "βλέπουμε" έναν υπολογιστή απλώς σαν ηλεκτρική συσκευή, είναι διαθέσιμες μόνο οι λειτουργίες που αυτός κληρονομεί από την κλάση ElectricDevice. Επομένως, λαμβάνοντας υπόψη τη δήλωση του betaX, η κλήση betaX.operate() θα παράγει λάθος μεταγλώττισης. Πράγματι, όταν ο μεταγλωττιστής θα συναντήσει αυτήν την κλήση, θα ελέγξει καταρχάς τι τύπου είναι η μεταβλητή betaX και θα δει πως είναι τύπου ElectricDevice. Στη συνέχεια θα ψάξει στην κλάση ElectricDevice να βρει μέθοδο με την ταυτότητα της operate. Εφόσον δεν θα την βρει, θα παράξει λάθος μεταγλώττισης.

Ωστόσο, εφόσον ο προγραμματιστής γνωρίζει πως στη μεταβλητή betaX τύπου ElectricDevice έχει εκχωρηθεί πραγματικό αντικείμενο τύπου Computer, μπορεί να προσπελάσει τις μεθόδους της κλάσης Computer με απλή μετατροπή τύπου, ((Computer)betaX).operate().

Δεν μπορούμε λοιπόν να καλέσουμε χωρίς μετατροπή τύπου τις μεθόδους της Computer από το στιγμιότυπο betaX. Μπορούμε όμως να καλέσουμε τις μεθόδους της γονικής κλάσης ElectricDevice, δηλαδή η κλήση betaX.setPowerOn(true) είναι απολύτως ορθή. Αυτή η δυνατότητα είναι πολύτιμη όπως θα διαπιστώσετε αμέσως παρακάτω.

Ας υποθέσουμε πως έχουμε και άλλους τύπους ηλεκτρικών συσκευών. Πιο συγκεκριμένα, στην εφαρμογή μας έχουμε μια κλάση που αναπαριστά τα πλυντήρια (WashingMachine), μια κλάση που αναπαριστά τους ηλεκτρικούς φούρνους (Oven) και μία κλάση που αναπαριστά τα κλιματιστικά (AirCondition). Όλες αυτές είναι ηλεκτρικές συσκευές. Στον κώδικα 13.9 δίνουμε τον ορισμό αυτών των κλάσεων:

```
public class AirCondition extends ElectricDevice {

    public void operate() {
        System.out.println("Το air condition δροσίζει τον χώρο");
    }
}
```

```
public class Oven extends ElectricDevice {

    public void operate() {
        System.out.println("Ο φούρνος ψήνει το φαγητό");
    }
}

public class WashingMachine extends ElectricDevice {

    public void operate() {
        System.out.println("Το πλυντηριο πλένει τα ρούχα");
    }
}
```

Κώδικας 13.9 *Τρεις κλάσεις παράγωγες της ElectricDevice*

Ας υποθέσουμε τώρα πως η εφαρμογή μας έχει να κάνει με το έξυπνο σπίτι στο οποίο θέλουμε να έχουμε έναν τρόπο να ενεργοποιούμε, να απενεργοποιούμε και να ελέγχουμε ταυτόχρονα όλες τις διαθέσιμες ηλεκτρικές συσκευές. Πώς μπορούμε να το πετύχουμε; Παρουσιάζουμε τον κατάλληλο κώδικα και στη συνέχεια τον επεξηγούμε.

```
public class SmartHome {

    public static void switchOn(ElectricDevice[] homeElectrics) {
        for (ElectricDevice eD : homeElectrics) {
            eD.setPowerOn(true);
        }
    }

    public static void switchOff(ElectricDevice[] homeElectrics) {
        for (ElectricDevice eD : homeElectrics) {
            eD.setPowerOn(false);
        }
    }

    public static boolean checkDevices(ElectricDevice[] homeElectrics) {
        for (ElectricDevice eD : homeElectrics) {
            if (!eD.isPowerOn()) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        ElectricDevice[] homeElectrics = new ElectricDevice[4];
        homeElectrics[0] = new WashingMachine();
        homeElectrics[1] = new AirCondition();
        homeElectrics[2] = new Oven();
        homeElectrics[3] = new Computer();
        switchOn(homeElectrics);
        System.out.println(checkDevices(homeElectrics));
        switchOff(homeElectrics);
        System.out.println(checkDevices(homeElectrics));
    }
}
```

Κώδικας 13.10 *Εφαρμογή 'Το έξυπνο σπίτι'.*

Ας συζητήσουμε τον κώδικα 13.10 ξεκινώντας από την main. Στην main έχουμε ορίσει έναν πίνακα για αντικείμενα ElectricDevice. Θα μπορούσαμε να έχουμε χρησιμοποιήσει κάποια άλλη κλάση, π.χ.

ArrayList<ElectricDevice>. Η επιλογή αυτή όμως δεν σχετίζεται με τις έννοιες που παρουσιάζουμε εδώ. Σε κάθε θέση του, ο πίνακας homeElectrics μπορεί να φιλοξενήσει μια μεταβλητή αναφοράς τύπου ElectricDevice και αφού είπαμε πως σε τέτοια μεταβλητή μπορούμε να εκχωρήσουμε αντικείμενα παράγωγων κλάσεων, εκχωρούμε τις συσκευές του έξυπνου σπιτιού. Επομένως, μπορούμε τώρα από τις αναφορές που έχουμε στον πίνακα να καλέσουμε μεθόδους της ElectricDevice. Αυτό ακριβώς κάνουμε στις συναρτήσεις switchOn, switchOff και checkDevices. Ο κώδικάς μας είναι πολυμορφικός. Η συνάρτηση switchOn για παράδειγμα επενεργεί σε μεταβλητές διαφορετικών τύπων, ενώ ταυτόχρονα διατηρούμε τον απαραίτητο έλεγχο τύπου. Ο πίνακας homeElectrics δέχεται μόνο αντικείμενα κάθε ένα εκ των οποίων είναι μια ηλεκτρική συσκευή. Ωστόσο, τα αντικείμενα αυτά εξειδικεύονται περαιτέρω ως προς τον τύπο τους. Για όλα όμως υπάρχει η εγγύηση ότι μπορούν να καλέσουν τις συναρτήσεις που έχουν οριστεί στη γονική κλάση.

Ας υποθέσουμε τώρα πως κάποιες μέθοδοι της ElectricDevice έχουν επανορισθεί στις παράγωγες. Έστω ότι τα αντικείμενα της κλάσης Computer πριν συνδεθούν στο ρεύμα, πρέπει να συνδεθούν σε κατάλληλο μετασχηματιστή. Αυτό μπορούμε να το πετύχουμε εύκολα προσθέτοντας τον κώδικα 13.11 στην κλάση Computer.

```
private void connectToTransformer() {
    System.out.println("connected");
}

@Override
public void setPowerOn(boolean powerOn) { //Κώδικας 13.11
    if (powerOn) {
        connectToTransformer();
    }
    super.setPowerOn(powerOn);
}
```

Κώδικας 13.11 *Προσθήκη λειτουργίας σύνδεσης σε μετασχηματιστή και επανορισμός της setPowerOn στην κλάση Computer.*

Το ερώτημα που τίθεται εδώ είναι ποια setPowerOn θα καλέσει ο κώδικας 13.10. Αυτήν που ορίσθηκε στην ElectricDevice ή αυτήν που ορίσαμε στην κλάση Computer. Είναι γεγονός πως πριν μετατρέψουμε την κλάση Computer, ο κώδικας καλεί την setPowerOn της ElectricDevice. Μετά όμως τη μετατροπή της κλάσης Computer χωρίς καμία επιπλέον μετατροπή, ο κώδικας 13.10 θα καλέσει την setPowerOn που επανορίστηκε στην κλάση Computer. Ισχύει λοιπόν ο γενικός κανόνας πως από μια αναφορά τύπου γονικής καλούνται μόνο συναρτήσεις για τις οποίες υπάρχει ορισμός στη γονική, ωστόσο καλείται η υλοποίηση της κλάσης που αντιστοιχεί στο πραγματικό αντικείμενο. Αν στην παράγωγη έχει επανοριστεί μια μέθοδος καλείται αλλιώς η κληρονομούμενη. Το χαρακτηριστικό αυτό ονομάζεται δυναμική δέσμευση (dynamic binding), καθώς η ακριβής ταυτότητα της μεθόδου που θα κληθεί υπολογίζεται κατά τον χρόνο εκτέλεσης και όχι κατά τον χρόνο μεταγλώττισης.

Αυτό το χαρακτηριστικό πολλαπλασιάζει την αξία της κληρονομικότητας και του πολυμορφισμού, γεγονός που θα έχουμε πολλές ευκαιρίες να διαπιστώσουμε στη συνέχεια.

13.2.1 Ο τελεστής instanceof

Όπως είδαμε σε μια μεταβλητή αναφοράς τύπου γονικής είναι εφικτό να εκχωρήσουμε αναφορά τύπου παράγωγης, δηλαδή η εκχώρηση

```
ElectricDevice eD = new WashingMachine();
```

είναι απολύτως σωστή. Αυτό σημαίνει πως μέσα στο πρόγραμμά μας είναι δυνατόν να έχουμε μεταβλητές τύπου γονικής που όμως αναφέρονται σε πραγματικά στιγμιότυπα παράγωγης, όπως ακριβώς η μεταβλητή eD. Είναι επίσης πιθανό, η παράγωγη κλάση να έχει ορίσει επιπλέον δικές της μεθόδους που δεν υφίστανται στη γονική. Για παράδειγμα, ας θεωρήσουμε πως η κλάση WashingMachine περιλαμβάνει μια adjustWaterTemperature, όπως δείχνει ο κώδικας 12.13.

```
public class WashingMachine extends ElectricDevice {
```

```
        double temperature = 20;

        public void adjustWaterTemperature(double temperature) {
            this.temperature = temperature;
        }

        @Override
        public void operate() {
            System.out.println("Το πλυντηριο πλένει τα ρούχα στους
    "+temperature+" βαθμούς");
        }
    }
```

Κώδικας 13.12 *Η κλάση WashingMachine εφοδιασμένη με την μέθοδο adjustWaterTemperature.*

Η μέθοδος adjustWaterTemperature δεν μπορεί να κληθεί από το αντικείμενο eD του οποίου ο τύπος είναι ElectricDevice. Ο μόνος τρόπος για να κληθεί είναι η μετατροπή τύπου του στιγμιότυπου eD από ElectricDevice σε WashingMachine (downcasting). Είναι όμως ασφαλής μια τέτοια μετατροπή; Κάτω από ποιες συνθήκες μπορούμε να είμαστε σίγουροι πως μια μεταβλητή τύπου ElectricDevices αναφέρεται σε πραγματικό αντικείμενο τύπου WashingMachine; Την απάντηση μας δίνει ο τελεστής instanceof με τον οποίο ελέγχουμε τον τύπο του πραγματικού αντικειμένου μιας μεταβλητής αναφοράς. Ο κώδικας 13.13 επιδεικνύει τη χρήση του τελεστή instanceof.

```
public class InstanceOfDemo {

    public static void main(String[] args) {

        ElectricDevice eD = new WashingMachine();
        eD.setPowerOn(true);
        if (eD instanceof WashingMachine) {
            ((WashingMachine) eD).adjustWaterTemperature(30);
        }
        eD.operate();
        System.out.println(eD instanceof ElectricDevice);
    }
}
```

Κώδικας 13.13 *Χρήση του τελεστή instanceof.*

Προσέξτε πως η έκφραση στην τρίτη γραμμή της main αλλά και η έξοδος της τελευταίας γραμμής της main είναι true. Πράγματι, η μεταβλητή eD είναι ένα πλυντήριο αλλά είναι και μια ηλεκτρική συσκευή. Ο τελεστής instanceof θα επιστρέψει true για κάθε κλάση με την οποία η εξεταζόμενη μεταβλητή σχετίζεται με τη σχέση ‘είναι ένα’.

13.3 Αφηρημένες κλάσεις

Θα παρατηρήσατε πως όλες οι κλάσεις ηλεκτρικών συσκευών που έχουμε ορίσει διαθέτουν μια μέθοδο operate. Πρόκειται για τη μέθοδο που τις θέτει σε λειτουργία. Στην εφαρμογή έξυπνο σπίτι θα θέλαμε να έχουμε έναν τρόπο να θέσουμε σε λειτουργία αυτομάτως όλες τις ηλεκτρικές συσκευές ενός σπιτιού. Δυστυχώς, έτσι όπως είναι ο κώδικάς μας κάτι τέτοιο δεν είναι εφικτό. Αν το επιχειρήσουμε, όπως εξηγήσαμε προηγουμένως, ο μεταγλωττιστής θα διαμαρτυρηθεί, καθώς δεν θα βρει μέθοδο operate στη γονική κλάση ElectricDevice. Βέβαια αυτό μπορούμε να το λύσουμε αν προσθέσουμε μια μέθοδο operate στην κλάση ElectricDevice. Τι να κάνει όμως μια τέτοια μέθοδος στην ElectricDevice; Η λειτουργία της κλάσης WashingMachine είναι να πλένει τα ρούχα, της Oven να ψήνει το φαγητό, της AirCondition να δροσίζει τον χώρο. Ποια είναι όμως η λειτουργία της ElectricDevice; Παρότι κάθε ηλεκτρική συσκευή επιτελεί κάποια λειτουργία, στο επίπεδο της ElectricDevice δεν μπορεί να καθοριστεί το περιεχόμενό της. Υπάρχει αλλά είναι ακαθόριστη. Με άλλα λόγια είναι αφηρημένη.

Πράγματι, το μοντέλο του αντικειμενοστρεφούς προγραμματισμού μας παρέχει τη δυνατότητα δήλωσης αφηρημένων (abstract) μεθόδων. Μία αφηρημένη μέθοδος, συμπεριλαμβάνει στη δήλωσή της τη λέξη-κλειδί abstract και δεν ακολουθείται από μπλοκ κώδικα, αντίθετα τερματίζεται από τον χαρακτήρα ';'. Προσθέστε στην κλάση ElectricDevive την αφηρημένη μέθοδο operate.

```
public abstract void operate();
```

Μία κλάση που περιλαμβάνει μία ή περισσότερες αφηρημένες μεθόδους, πρέπει και η ίδια να οριστεί ως αφηρημένη. Μετατρέψτε την επικεφαλίδα της κλάσης ElectricDevice ως ακολούθως:

```
public abstract class ElectricDevice {…}
```

Αντίθετα, μια κλάση μπορεί να οριστεί να είναι αφηρημένη ακόμη και αν καμία μέθοδός της δεν είναι αφηρημένη. Ποιο είναι όμως ακριβώς το νόημα της λέξης-κλειδί abstract;

Ο μεταγλωττιστής δεν επιτρέπει τη δημιουργία αντικειμένων μιας abstract κλάσης. Αν υποθέσουμε πως είχαμε τη δυνατότητα να δημιουργήσουμε ένα αντικείμενο της αφηρημένης έκδοσης της ElectricDevice, τι θα συνέβαινε αν το αντικείμενο καλούσε την abstract μέθοδο operate; Είναι προφανές πως η εφαρμογή μας θα οδηγείτο σε αδιέξοδο με την κλήση μιας μεθόδου για την οποία δεν έχει οριστεί σώμα.

Οι παράγωγες κλάσεις της ElectricDevice κληρονομούν και τις αφηρημένες μεθόδους. Αν τις ορίσουν, τότε είναι σαφείς (concrete) κλάσεις, δηλαδή κλάσεις που παρέχουν ορισμό όλων των μεθόδων τους και για τις οποίες μπορούμε να δημιουργήσουμε αντικείμενα. Αν δεν οριστούν οι κληρονομούμενες αφηρημένες μέθοδοι στην παράγωγη κλάση, τότε παραμένει και αυτή αφηρημένη.

Οι αφηρημένες κλάσεις και μέθοδοι ενισχύουν την ισχύ του πολυμορφισμού. Έχοντας ορίσει την αφηρημένη μέθοδο operate στην ElectricDevice εγγυόμαστε στην ουσία πως κάθε απόγονός της διαθέτει αυτήν τη μέθοδο. Επομένως, κάθε αντικείμενο που μπορεί να δημιουργηθεί τύπου κλάσης απογόνου της ElectricDevice μπορεί με ασφάλεια να καλέσει την operate.

Έτσι μετά τη μετατροπή της ElectricDevice σε abstract, η εφαρμογή έξυπνο σπίτι μπορεί να τροποποιηθεί ώστε να ενεργοποιεί αυτόματα την operate για όλες τις ηλεκτρικές συσκευές του σπιτιού. Στον κώδικα 13.14 δίνεται η νέα έκδοση της SmartHome:

```
public class SmartHome {

    public static void switchOn(ElectricDevice[] homeElectrics) {
        for (ElectricDevice eD : homeElectrics) {
            eD.setPowerOn(true);
        }
    }

    public static void switchOff(ElectricDevice[] homeElectrics) {
        for (ElectricDevice eD : homeElectrics) {
            eD.setPowerOn(false);
        }
    }

    public static boolean checkDevices(ElectricDevice[] homeElectrics) {
        for (ElectricDevice eD : homeElectrics) {
            if (!eD.isPowerOn()) {
                return false;
            }
        }
        return true;
    }

    public static void operate(ElectricDevice[] homeElectrics) {
        for (ElectricDevice eD : homeElectrics) {
            eD.operate();
        }
    }
```

```
    public static void main(String[] args) {
        ElectricDevice[] homeElectrics = new ElectricDevice[4];
        homeElectrics[0] = new WashingMachine();
        homeElectrics[1] = new AirCondition();
        homeElectrics[2] = new Oven();
        homeElectrics[3] = new Computer();
        switchOn(homeElectrics);
        operate(homeElectrics);
        switchOff(homeElectrics);
        System.out.println(checkDevices(homeElectrics));
    }
}
```

Κώδικας 13.14 *Έκδοση της SmartHome που υποστηρίζει τη λειτουργία operate για όλες τις παράγωγες της ElectricDevice.*

13.4 Λυμένες Ασκήσεις

13.4.1 Γεωμετρικά σχήματα

Να υλοποιηθεί εφαρμογή που υπολογίζει το εμβαδόν μιας σειράς γεωμετρικών σχημάτων. Πιο συγκεκριμένα, η εφαρμογή να υποστηρίζει τον υπολογισμό του εμβαδού, ορθογώνιων παραλληλόγραμμων, τετραγώνων, κύκλων και τριγώνων. Για τα ορθογώνια, γνωρίζουμε το μήκος και το πλάτος, για τους κύκλους την ακτίνα και για τα τρίγωνα τη βάση και το ύψος. Τα τετράγωνα αναπαρίστανται σαν ορθογώνια με ίδιο μήκος και πλάτος.
Η εφαρμογή θα πρέπει να φορτώσει σε μία λίστα δύο γεωμετρικά σχήματα από κάθε είδος, δηλαδή σύνολο οκτώ και στη συνέχεια να εμφανίσει την ταυτότητα κάθε σχήματος συνοδευόμενη από το εμβαδόν του, όπως δείχνει το υπόδειγμα που ακολουθεί.

```
Κύκλος ακτίνας 4.5, εμβαδόν: 28.274
Κύκλος ακτίνας 6.0, εμβαδόν: 37.699
Ορθογώνιο μήκους 3.8, πλάτους 5.0, εμβαδόν: 19
Ορθογώνιο μήκους 6.0, πλάτους 2.6, εμβαδόν: 15.6
Τετράγωνο πλευράς 9.0, εμβαδόν: 81
Τετράγωνο πλευράς 3.0, εμβαδόν: 9
Τρίγωνο βάσης 4.4, ύψους=2.0, εμβαδόν: 4.4
Τρίγωνο βάσης 4.4, ύψους=2.2, εμβαδόν: 4.84
```

Το εμβαδόν των σχημάτων θα πρέπει να εμφανίζεται με ακρίβεια τριών δεκαδικών.

Λύση
Καταρχάς, προσέξτε πως σύμφωνα με την εκφώνηση, θα πρέπει οκτώ γεωμετρικά σχήματα να φορτωθούν σε ένα ArrayList. Επομένως, χρειαζόμαστε μια κλάση που αναπαριστά τα γεωμετρικά σχήματα και η οποία θα είναι γονική για τις κλάσεις των ορθογώνιων, των κύκλων και των τριγώνων. Επειδή η κοινή λειτουργία δεν είναι παρά ο υπολογισμός του εμβαδού, η κλάση μπορεί να διαθέτει μόνο μία μέθοδο. Βεβαίως, ο υπολογισμός του εμβαδού είναι εντελώς διαφορετικός για τα ορθογώνια, τους κύκλους και τα τρίγωνα. Παρότι, γνωρίζουμε πως κάθε γεωμετρικό σχήμα από αυτά που μας ενδιαφέρουν έχει εμβαδόν, το εμβαδόν δεν ορίζεται στο επίπεδο του γεωμετρικού σχήματος. Αυτή είναι ιδανική περίπτωση για να κάνουμε την μέθοδο υπολογισμού του εμβαδού αφηρημένη στην κλάση των γεωμετρικών σχημάτων. Ακολουθεί ο κώδικας της κλάσης των γεωμετρικών σχημάτων:

```
public abstract class Shape {
    public abstract double getArea();
}
```

Κώδικας 13.15 *Η αφηρημένη κλάση των γεωμετρικών σχημάτων.*

Στη συνέχεια θα πρέπει να δημιουργήσουμε σαφείς κλάσεις παράγωγες της Shape για κάθε έναν επιμέρους τύπο γεωμετρικού σχήματος.

```
public class Circle extends Shape {

    private double r;

    public Circle(double r) {
        this.r = r;
    }

    @Override
    public double getArea() {
        return 2 * Math.PI * r;
    }

    @Override
    public String toString() {
        return "Κύκλος ακτίνας " + r;
    }
}
```

Κώδικας 13.16 *Η παράγωγη κλάση Circle.*

Παραθέτουμε εδώ μόνο τις μεθόδους που είναι αναγκαίες για την εφαρμογή. Αναγνώστες, ρυθμιστές και οι μέθοδοι equals και hashCode παραλείπονται για απλοποίηση της παρουσίασης των εννοιών της κληρονομικότητας και του πολυμορφισμού. Η κλάση Circle είναι παράγωγη της Shape, διαθέτει έναν δημιουργό που δίνει τιμή στην ακτίνα του κύκλου, υλοποιεί την αφηρημένη μέθοδο getArea της Shape και επανορίζει τη μέθοδο toString.
Παρόμοια είναι και η υλοποίηση της κλάσης των ορθογώνιων.

```
public class Rectangle extends Shape {

    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public double getArea() {
        return length * width;
    }

    @Override
    public String toString() {
        return "Ορθογώνιο μήκους " + length + ", πλάτους " + width;
    }

    public double getLength() {
        return length;
    }

    public double getWidth() {
        return width;
    }
}
```

Κώδικας 13.17 *Η παράγωγη κλάση Rectangle.*

Εδώ έχουμε προσθέσει και τους αναγνώστες καθώς όπως βλέπουμε αμέσως παρακάτω είναι χρήσιμοι (ο ένας από τους δύο) στην κλάση των τετραγώνων.
Η κλάση των τετραγώνων μπορεί εύκολα να οριστεί ως παράγωγη της κλάσης Rectangle.

```
public class Square extends Rectangle {

    public Square(double length) {
        super(length, length);
    }

    @Override
    public String toString() {
        return "Τετράνωνο πλευράς " + getLength();
    }
}
```

Κώδικας 13.18 *Η κλάση Square παράγωγη της Rectangle.*

Η κλάση των τετραγώνων είναι παράγωγη της Rectangle, επομένως είναι παράγωγη και της Shape. Ο δημιουργός της κλάσης καλεί τον δημιουργό της γονικής όπου περνάει μήκος ίσο με πλάτος, μια και ένα τετράγωνο δεν είναι παρά ένα ορθογώνιο παραλληλόγραμμο με ίσο μήκος και πλάτος. Η μέθοδος getArea δεν επανορίζεται εδώ καθώς η κληρονομούμενη από την Rectangle είναι ικανοποιητική.
Τέλος, ακολουθεί η κλάση των τριγώνων:

```
public class Triangle extends Shape {

    private final double base;
    private final double high;

    public Triangle(double base, double high) {
        this.base = base;
        this.high = high;
    }

    @Override
    public double getArea() {
        return base * high / 2;
    }

    @Override
    public String toString() {
        return "Τρίγωνο βάσης " + base + ", ύψους=" + high;
    }
}
```

Κώδικας 13.19 *Η παράγωγη κλάση Triangle.*

Τώρα είμαστε έτοιμοι για να αναπτύξουμε την εφαρμογή σύμφωνα με τις απαιτήσεις της άσκησης.

```
import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;
import java.util.ArrayList;

public class CalcAreas {

    public static void main(String[] args) {//Κώδικας 13.20
        ArrayList<Shape> shapes = new ArrayList<>();
        shapes.add(new Circle(4.5));
        shapes.add(new Circle(6));
        shapes.add(new Rectangle(3.8, 5));
        shapes.add(new Rectangle(6, 2.6));
```

```
            shapes.add(new Square(9));
            shapes.add(new Square(3));
            shapes.add(new Triangle(4.4, 2));
            shapes.add(new Triangle(4.4, 2.2));
            DecimalFormatSymbols dfs = new DecimalFormatSymbols();
            dfs.setDecimalSeparator('.');
            dfs.setGroupingSeparator(',');
            DecimalFormat f = new DecimalFormat("######.###");
            f.setDecimalFormatSymbols(dfs);

            for (Shape shape : shapes) {
                System.out.println(shape + ", εμβαδόν: " +
    f.format(shape.getArea()));
            }
        }
    }
```

Κώδικας 13.20 *Η εφαρμογή CalcAreas.*

Σύμφωνα με την εκφώνηση, η εφαρμογή μας φορτώνει οκτώ γεωμετρικά σχήματα σε ένα ArrayList και στη συνέχεια εμφανίζει την String αναπαράσταση και το εμβαδόν καθενός. Παρατηρήστε πως κατά τη διαμόρφωση της εξόδου καθορίζουμε το διαχωριστικό δεκαδικών να είναι ο χαρακτήρας '.' και χιλιάδων να είναι ο χαρακτήρας ','.

13.4.2 Η κλάση Quiz

Να υλοποιήσετε την κλάση Quiz που διαθέτει τις ακόλουθες λειτουργίες:

1. Παρέχει λειτουργία πρόσθεσης ερωτήσεων.
2. Υποστηρίζει ερωτήσεις σωστού-λάθους και πολλαπλών επιλογών
3. Κατά την εκτέλεσή της, υποβάλλει τον χρήστη σε τεστ με βάση τις καταχωρημένες ερωτήσεις.
4. Για κάθε ερώτηση επιβεβαιώνει τη σωστή απάντηση ή αν ο χρήστης απαντήσει λανθασμένα, εμφανίζει τη σωστή απάντηση.

Λύση

Από την εκφώνηση είναι φανερό πως έχουμε να διαχειριστούμε ερωτήσεις δύο τύπων, σωστού-λάθους και πολλαπλών επιλογών. Και οι δύο τύποι όμως, παρότι εξειδικεύονται, παραμένουν ερωτήσεις. Επομένως, ένας σχεδιασμός με γονική κλάση, αυτή των ερωτήσεων και παράγωγες τις κλάσεις των ερωτήσεων σωστού-λάθους και πολλαπλών επιλογών θα φανεί χρήσιμος.

Οι λειτουργίες που είναι αναγκαίο να δηλωθούν στη γονική κλάση είναι η λήψη του κειμένου της ερώτησης και η αξιολόγηση της απάντησης. Οι λειτουργίες αυτές μπορούν να υλοποιηθούν στις παράγωγες, όχι όμως και στη γονική. Ως εκ τούτου, η γονική θα είναι αφηρημένη.

```
public abstract class Question {
    public abstract String getQuestion();
    public abstract boolean evaluateAnswer(String answer);
}
```

Κώδικας 13.21 *Η αφηρημένη κλάση Question.*

Στη συνέχεια, στον κώδικα 13.22, παρουσιάζεται η υλοποίηση της κλάσης των ερωτήσεων σωστού-λάθους.

```
public class TrueFalse extends Question {
    String text;
    boolean correctAnswer;

    public TrueFalse(String text, boolean answer) {
        this.text = text;
        this.correctAnswer = answer;
    }
```

```
    @Override
    public String getQuestion() {
        return text +"\n t(rue) or f(alse)\n";
    }

    @Override
    public boolean evaluateAnswer(String userAnswer) {
        return
String.valueOf(correctAnswer).toLowerCase().charAt(0)==userAnswer.toLowerCa
se().charAt(0);
    }
}
```

Κώδικας 13.22 *Η κλάση των ερωτήσεων true-false.*

Ο δημιουργός της κλάσης μας επιτρέπει να καθορίσουμε το κείμενο της ερώτησης και να προσδιορίσουμε ποια απάντηση είναι σωστή. Η getQuestion επιστρέφει μια συμβολοσειρά που αποτελείται από το κείμενο της ερώτησης ακολουθούμενης από τις επιλογές true ή false στην επόμενη από το κείμενο γραμμή. Η evaluateAnswer ελέγχει ώστε ο χαρακτήρας που έδωσε ως απάντηση ο χρήστης να είναι ίδιος με τον χαρακτήρα από τον οποίο αρχίζει η ορθή απάντηση της ερώτησης. Και οι δύο χαρακτήρες μετατρέπονται σε πεζούς ώστε η σύγκριση να γίνει αγνοώντας τυχόν διαφορές μεταξύ πεζών και κεφαλαίων.
Ο κώδικας 13.23 παρουσιάζει την κλάση ερωτήσεων πολλαπλής επιλογής.

```
import java.util.ArrayList;
import java.util.Arrays;

public class MultipleChoice extends Question {

    private final String text;
    private final ArrayList<String> answers;
    private final String correctAnswer;

    public MultipleChoice(String text,String correctAnswer,
String...answers) {
        this.text = text;
        this.answers=new ArrayList<>(Arrays.asList(answers));
        this.correctAnswer=correctAnswer;
    }

    @Override
    public String getQuestion() {
        StringBuilder rVal=new StringBuilder();
        rVal.append(text+"\n");
        char label='A';
        for (String answer:answers) {
            rVal.append(label).append(". ").append(answer).append("\n");
            label++;
        }
        return rVal.toString();
    }

    @Override
    public boolean evaluateAnswer(String userAnswer) {
        return correctAnswer.equalsIgnoreCase(userAnswer);
    }

    public String getCorrectAnswer() {
        return correctAnswer;
    }
}
```

Κώδικας 13.23 *Η κλάση των ερωτήσεων πολλαπλής επιλογής.*

Ο δημιουργός της κλάσης λαμβάνει ως παράμετρο το κείμενο της ερώτησης, την αρίθμηση της ορθής επιλογής και τις εναλλακτικές επιλογές. Προκειμένου να υπάρχει η δυνατότητα δημιουργίας ερωτήσεων με διαφορετικό πλήθος δυνητικών απαντήσεων, ο δημιουργός διαθέτει λίστα παραμέτρων μεταβλητού μήκους. Η getQuestion αριθμεί τις εναλλακτικές επιλογές αυτόματα με κεφαλαίους λατινικούς χαρακτήρες αρχίζοντας από τον χαρακτήρα 'A' προς τον 'Z'. Επομένως, αν ο χρήστης της κλάσης δημιουργήσει ένα αντικείμενο MultipleChoice και τοποθετήσει πρώτη κατά σειρά τη σωστή απάντηση, στην παράμετρο του δημιουργού correctAnswer θα πρέπει να δώσει τον χαρακτήρα 'A', αν τοποθετήσει δεύτερη τη σωστή απάντηση θα πρέπει να δώσει τον χαρακτήρα 'B', κοκ. Ο αναγνώστης της σωστής απάντησης είναι απαραίτητος ώστε τα τεστ που χρησιμοποιούν τέτοιες ερωτήσεις να είναι σε θέση να πληροφορήσουν σχετικά τον χρήστη, όπως φαίνεται στον κώδικα 13.24 που ακολουθεί.

Έχοντας τις κλάσεις Question, TrueFalse και MultipleChoice είναι εύκολο να κατασκευάσουμε τεστ που βασίζονται σε αυτούς τους τύπους ερωτήσεων. Ο κώδικας 13.24 παρουσιάζει μία τέτοια κλάση που είναι σύμφωνη με τις απαιτήσεις της εκφώνησης.

```
import java.util.ArrayList;
import java.util.Scanner;

public class Quiz {

    static ArrayList<Question> test = new ArrayList<>();

    private static void setUpQuestions() {
        test.add(new TrueFalse("Η γάτα είναι κατοικίδιο", true));
        test.add(new TrueFalse("Ο σκύλος δεν εξημερώνεται", false));
        test.add(new MultipleChoice("Ο καλύτερος φίλος του ανθρώπου είναι ο",
"C", "όφις", "άλογο", "σκύλος"));
        test.add(new MultipleChoice("Η for είναι", "B", "δομή ελέγχου",
"επαναληπτική διαδικασία", "αναδρομική συνάρτηση"));
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        setUpQuestions();
        for (Question q : test) {
            System.out.print(q.getQuestion());
            System.out.print("Εισάγετε την απάντηση: ");
            String userAnswer = in.next();
            if (q.evaluateAnswer(userAnswer)) {
                System.out.println("Σωστή απάντηση");
            } else if (q instanceof MultipleChoice) {
                System.out.println("Λάθος Απάντηση. Σωστή απάντηση η " +
((MultipleChoice) q).getCorrectAnswer());
            } else {
                System.out.println("Λάθος Απάντηση");
            }
            System.out.println();
        }

    }
}
```

Κώδικας 13.24 *Η κλάση των Quiz.*

Στην main καλείται καταρχάς η setUpQuestions που δημιουργεί και φορτώνει στο ArrayList test τις ερωτήσεις πολλαπλής επιλογής ή σωστού-λάθους. Στη συνέχεια κάθε μια ερώτηση τίθεται στον χρήστη με τη βοήθεια της getQuestion, λαμβάνεται η απάντηση του χρήστη η οποία αξιολογείται από την evaluateAnswer.

13.4.3 Μισθοδοσία Μηνός

Η εταιρεία CompSystems απασχολεί τριών τύπων εργαζόμενους, τους υπάλληλους γραφείου, τα διευθυντικά στελέχη και τους συμβασιούχους. Όλοι οι εργαζόμενοι στην εταιρεία είναι φυσικά πρόσωπα. Ως τέτοια κληρονομούν όλα τα χαρακτηριστικά της κλάσης Person (κώδικας 12.19). Οι αποδοχές των υπαλλήλων γραφείου βασίζονται σε σταθερό μισθό 1200 Ευρώ τον μήνα. Επιπλέον, κάθε υπάλληλος γραφείου λαμβάνει μηνιαία ένα ποσό που υπολογίζεται ως το γινόμενο 5% x Αριθμός ετών στην εταιρεία x σταθερός μισθός.

Τα διευθυντικά στελέχη αμείβονται με σταθερό μισθό 3500 Ευρώ συν ένα ποσό που λαμβάνουν επίσης μηνιαία και έχει συμφωνηθεί μεταξύ εταιρείας και κάθε στελέχους ξεχωριστά.

Οι συμβασιούχοι αμείβονται με ένα ποσό ανά ώρα εργασίας που συμφωνείται μεταξύ της εταιρείας και κάθε συμβασιούχου ξεχωριστά. Κατά την έκδοση της μισθοδοσίας μηνός, οι συμβασιούχοι ενημερώνουν για τον αριθμό ωρών που εργάστηκαν κατά τη διάρκεια του μήνα.

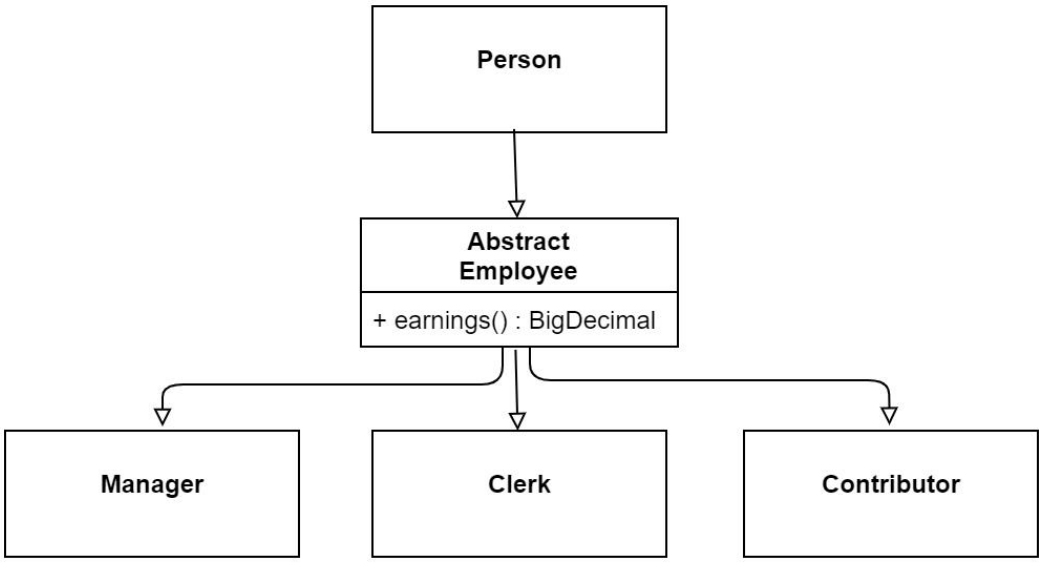
Η εταιρεία διαθέτει τρεις υπαλλήλους γραφείου, δύο διευθυντές και δύο συμβασιούχους. Οι διευθυντές έχουν συμφωνήσει ως επιπρόσθετο μηνιαίο ποσό 2000 Ευρώ ο ένας και 2200 ο άλλος. Οι συμβασιούχοι έχουν συμφωνήσει ωρομίσθιο 12 Ευρώ ο πρώτος και 20 ο δεύτερος. Ο πρώτος εργάστηκε 120 ώρες και ο δεύτερος 140 ώρες αυτόν τον μήνα. Τέλος, οι υπάλληλοι έχουν προσληφθεί σε τυχαίες ημερομηνίες μέσα στις 10.000 προηγούμενες ημερομηνίες από την τρέχουσα.

Η εταιρεία θέλει να υλοποιήσετε εφαρμογή που υπολογίζει και εμφανίζει κατάσταση μισθοδοσίας του τρέχοντος μήνα. Η κατάσταση σε κάθε γραμμή θα πρέπει να περιλαμβάνει τον αριθμό ταυτότητας του υπαλλήλου, το ονοματεπώνυμό του και τις αποδοχές του μήνα.

Λύση

Όπως διατυπώνεται στην εκφώνηση, οι εργαζόμενοι στην εταιρεία είναι φυσικά πρόσωπα. Ωστόσο, ως εργαζόμενοι στην εταιρεία πρέπει να είναι σε θέση να υπολογίζουν το μηνιαίο εισόδημά τους. Επομένως, επιτελούν μια επιπλέον λειτουργία, δηλαδή μια λειτουργία που δεν συναντάται σε πρόσωπα που δεν είναι εργαζόμενοι. Από την άλλη πλευρά, ο υπολογισμός του μηνιαίου εισοδήματος βασίζεται σε εντελώς διαφορετικά δεδομένα και είναι γενικότερα διαφορετικός για κάθε τύπο εργαζόμενου.

Με αυτά τα δεδομένα, μας εξυπηρετεί η οργάνωση των κλάσεων, όπως φαίνεται στο σχήμα 13.3.



Σχήμα 13.3 *Διάγραμμα κλάσεων της εφαρμογής Μισθοδοσία Μηνός*

Ο εργαζόμενος αναπαρίσταται από την κλάση Employee, παράγωγη κλάση της Person. Ωστόσο, πρόκειται για αφηρημένη κλάση, καθώς δηλώνει τη συνάρτηση earnings που επιστρέφει το μηνιαίο εισόδημα του εργαζόμενου. Η συνάρτηση αυτή όμως δεν μπορεί να οριστεί σαφώς στο επίπεδο του εργαζόμενου. Μπορεί όμως να οριστεί στις παράγωγες κλάσεις της Employee που αναπαριστούν εργαζόμενους συγκεκριμένου τύπου. Η κλάση Manager αναπαριστά τους διευθυντές, η κλάση Clerk τους υπαλλήλους γραφείου, η κλάση Contributor τους συμβασιούχους.
Ας δούμε καταρχάς την ανάπτυξη των κλάσεων και στη συνέχεια την εφαρμογή μισθοδοσία μηνός.

```
import java.math.BigDecimal;
import java.time.LocalDate;

public abstract class Employee extends Person {

    public Employee(String id, String fName, String sName, LocalDate birthday) {
        super(id, fName, sName, birthday);
    }

    public abstract BigDecimal earnings();
}
```

Κώδικας 13.25 *Η αφηρημένη κλάση Employee*

Η κλάση Employee έχει ένα ιδιαίτερο χαρακτηριστικό. Κληρονομεί την κλάση Person και ταυτόχρονα είναι αφηρημένη. Παρότι δεν είναι εφικτή η δημιουργία αντικειμένων της κλάσης, απαιτείται να οριστεί δημιουργός ο οποίος εκείνο που κάνει είναι ότι καλεί τον δημιουργό της γονικής Person. Ο δημιουργός εδώ είναι απαραίτητος ώστε οι παράγωγες κλάσεις Manager, Clerk και Contributor να μπορούν να τον καλέσουν και αυτός με τη σειρά του να καλέσει τον δημιουργό Person. Κατά τα λοιπά, η κλάση δηλώνει την αφηρημένη συνάρτηση earnings που υπολογίζει το μηνιαίο εισόδημα του εργαζόμενου.

```
import java.math.BigDecimal;
import java.time.LocalDate;

public class Manager extends Employee {
    private final BigDecimal constantFee=new BigDecimal(3500);
    private final BigDecimal bonus;

    public Manager(BigDecimal bonus, String id, String fName, String sName,
LocalDate birthday) {
        super(id, fName, sName, birthday);
        this.bonus = bonus;
    }

    @Override
    public BigDecimal earnings() {
        return constantFee.add(bonus);
    }
}
```

Κώδικας 13.26 *Η κλάση Manager*

Ο δημιουργός της κλάσης Manager καλεί τον δημιουργό της Employee και στη συνέχεια δίνει τιμή στην πρόσθετη αμοιβή του στελέχους. Σύμφωνα με την εκφώνηση, η πρόσθετη αμοιβή καθορίζεται σε συμφωνία με την εταιρεία ξεχωριστά για κάθε στέλεχος. Επομένως, μπορεί η τιμή αυτή να καθοριστεί κατά τη δημιουργία των αντικειμένων Manager, ειδικά για κάθε αντικείμενο.

Στη συνέχεια, υλοποιείται η συνάρτηση earnings που απλώς επιστρέφει το άθροισμα του μηνιαίου μισθού συν την πρόσθετη αμοιβή.

Η κλάση Clerk παρουσιάζεται στον κώδικα 13.27:

```
import java.math.BigDecimal;
import java.time.LocalDate;
```

```
import java.time.Period;
import java.util.Random;

public class Clerk extends Employee {
    private static final BigDecimal BASESALARY=new BigDecimal(1200);
    private static final double INCREMENTRATE=0.05;
    private final LocalDate hireDate;

    public Clerk(String id, String fName, String sName, LocalDate birthday)
{
        super(id, fName, sName, birthday);
        hireDate=LocalDate.now().minusDays(new Random().nextInt(10000));
    }


    @Override
    public BigDecimal earnings() {
        LocalDate today = LocalDate.now();
        int workingYears=Period.between(hireDate, today).getYears();
        BigDecimal incrementAmount=BASESALARY.multiply(new
BigDecimal(workingYears*INCREMENTRATE));
        double d=incrementAmount.doubleValue();
        BigDecimal salary=BASESALARY.add(incrementAmount);
        return salary;
    }
}
```

Κώδικας 13.27 *Η κλάση Clerk*

Οι υπάλληλοι γραφείου αμείβονται με σταθερό μηνιαίο μισθό συν μια προσαύξηση που εξαρτάται από ένα σταθερό ποσοστό και από τα έτη προϋπηρεσίας στην εταιρεία. Εφόσον ο μηνιαίος μισθός και το ποσοστό προσαύξησης είναι σταθερά ποσά, οι μεταβλητές BASESALARY και INCREMENTRATE δηλώθηκαν ως final. Επιπλέον, τόσο ο μηνιαίος μισθός όσο και το ποσοστό προσαύξησης είναι κοινά για όλους τους υπαλλήλους γραφείου, επομένως, οι αντίστοιχες μεταβλητές θα πρέπει να δηλωθούν ως static, καθώς αφορούν την κλάση και όχι κάθε αντικείμενό της χωριστά. Η ημερομηνία πρόσληψης όμως αφορά κάθε εργαζόμενο χωριστά και δηλώνεται ως μεταβλητή στιγμιότυπου. Καθώς και αυτή δεν είναι λογικό να αλλάξει μετά την πρόσληψη εργαζόμενου, μπορεί επίσης να δηλωθεί ως final.

Ο δημιουργός της κλάσης, αφού καλέσει τον δημιουργό της γονικής, δίνει τιμή στην ημερομηνία πρόσληψης σύμφωνα με τις απαιτήσεις της εκφώνησης.

Στη συνέχεια, η κλάση επανορίζει τη συνάρτηση earnings. Υπολογίζει τα έτη από την ημερομηνία πρόσληψης μέχρι σήμερα, μετά, το ποσό προσαύξησης, το οποίο προσθέτει στον βασικό μισθό και επιστρέφει τις αποδοχές του μήνα.

Τέλος, οι συμβασιούχοι αναπαρίστανται από την κλάση Contributor (κώδικας 13.28):

```
import java.math.BigDecimal;
import java.time.LocalDate;

public class Contributor extends Employee {

    private int workingHours;
    private BigDecimal paymentPerHour;

    public Contributor(BigDecimal paymentPerHour, String id, String fName,
String sName, LocalDate birthday) {
        super(id, fName, sName, birthday);
        this.paymentPerHour = paymentPerHour;
    }

    public void setWorkingHours(int workingHours) {
        this.workingHours = workingHours;
```

```
    }

    @Override
    public BigDecimal earnings() {
        return paymentPerHour.multiply(new BigDecimal(workingHours));
    }
}
```

Κώδικας 13.28 *Η κλάση Contributor*

Οι συμβασιούχοι αμείβονται με βάση μια αμοιβή για την ώρα εργασίας τους που συμφωνείται μεταξύ εταιρείας και κάθε συμβασιούχου χωριστά. Η πληροφορία συνοδεύει κάθε συμβασιούχο από τη δημιουργία του και επομένως μπορεί να δοθεί στον δημιουργό. Οι ώρες εργασίας όμως αφορούν τις ώρες εργασίας του μήνα, πιθανότατα παρουσιάζουν διακυμάνσεις από μήνα σε μήνα και άρα δεν σχετίζονται με τη δημιουργία του αντικειμένου. Έτσι παρέχουμε τον ρυθμιστή setWorkingHours ώστε να ενημερώνεται κατάλληλα η αντίστοιχη μεταβλητή. Τέλος, η earnings επιστρέφει το γινόμενο ώρες εργασίας επί αμοιβή ανά ώρα.

Έχοντας διαθέσιμες τις κλάσεις, η ανάπτυξη της εφαρμογής είναι σχετικά εύκολη.

```
import java.math.BigDecimal;
import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;
import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;

public class CalcMonthlyPayment { //Κώδικας 13.29

    public static void main(String[] args) {
        ArrayList<Employee> personel = new ArrayList<>();
        personel.add(new Clerk("1", "Jim", "Black", LocalDate.of(1967,
Month.MARCH, 1)));
        personel.add(new Clerk("2", "John", "Blue", LocalDate.of(1972,
Month.APRIL, 12)));
        personel.add(new Clerk("3", "Jack", "Dalton", LocalDate.of(1980,
Month.MARCH, 21)));
        personel.add(new Manager(BigDecimal.valueOf(2000), "101", "Adam",
"Smith", LocalDate.of(1980, Month.JANUARY, 12)));
        personel.add(new Manager(BigDecimal.valueOf(2200), "102", "Adam",
"Smith", LocalDate.of(1980, Month.MARCH, 21)));
        Contributor cntr = new Contributor(BigDecimal.valueOf(12), "1001",
"Kalamity", "Jane", LocalDate.of(2000, Month.MARCH, 21));
        cntr.setWorkingHours(120);
        personel.add(cntr);
        cntr = new Contributor(BigDecimal.valueOf(20), "1002", "Erick",
"Erickson", LocalDate.of(2001, Month.MARCH, 21));
        cntr.setWorkingHours(140);
        personel.add(cntr);

        DecimalFormatSymbols dfs = new DecimalFormatSymbols();
        dfs.setDecimalSeparator('.');
        dfs.setGroupingSeparator(',');
        DecimalFormat f = new DecimalFormat("######.###");
        f.setDecimalFormatSymbols(dfs);

        for (Employee e : personel) {
            System.out.println(e.getId() + " " + e.getfName() + " " +
e.getsName() + " " + f.format(e.earnings().doubleValue()));
        }
    }
}
```

Κώδικας 13.29 *Η εφαρμογή CalcMonthlyPayment*

Καταρχάς, η εφαρμογή δημιουργεί ένα ArrayList για Employee. Στη συνέχεια δημιουργεί ένα αντικείμενο για κάθε εργαζόμενο και το προσθέτει στο ArrayList. Ειδικά για τα αντικείμενα Contributor πριν τα προσθέσει στο ArrayList, ενημερώνει τις ώρες εργασίας. Στη συνέχεια, φροντίζει για την κατάλληλη διαμόρφωση της εξόδου και μετά τυπώνει την κατάσταση μισθοδοσίας βασιζόμενη απλώς στις συναρτήσεις των κλάσεων. Οι getId, getfName και getsName έχουν κληρονομηθεί αυτούσιες από την κλάση Person. Η earnings, όπως είδαμε, ορίζεται στις σαφείς κλάσεις εργαζόμενων. Προκειμένου να αξιοποιήσουμε την format, μετατρέπουμε τα earnings από BigDecimal σε double.

13.5 Ασκήσεις προς Λύση

13.5.1 Ο ρόμβος

Στην άσκηση 13.4.1 υπολογισμού εμβαδού γεωμετρικών σχημάτων να υποστηριχτεί ο υπολογισμός εμβαδού του ρόμβου. Ο υπολογισμός του εμβαδού ρόμβου μπορεί να γίνει συνάρτηση των μηκών των διαγώνιων του σύμφωνα με τον τύπο που ακολουθεί

$$E = \frac{(\delta 1 \times \delta 2)}{2}, \delta 1 \; και \; \delta 2 \; οι \; διαγώνιοι$$

Επιπλέον, προσθέστε στις κλάσεις της άσκησης όλες τις αναγκαίες μεθόδους που έχουν παραληφθεί, π.χ. equals, hashCode, toString, ρυθμιστές και αναγνώστες.

13.5.2 Ζωολογικός κήπος

Σε έναν ζωολογικό κήπο έχουν συγκεντρωθεί διάφορα ζώα. Από αυτά, κάποια περπατούν, κάποια πετούν, κάποια κολυμπούν και κάποια έρπουν. Κατασκευάστε έναν ζωολογικό κήπο, προσθέστε ζώα από κάθε τύπο και θέστε τα σε κίνηση. Οι συναρτήσεις κίνησης θα βγάζουν μηνύματα που απλώς περιγράφουν την κίνηση, π.χ. το λιοντάρι περπατά, το δελφίνι κολυμπά.

13.5.3 Η συμπεριφορά των μεταβλητών αναφοράς

Στην άσκηση 13.4.3 ενημερώνουμε πρώτα τις ώρες εργασίας ενός αντικειμένου τύπου Contributor και στη συνέχεια προσθέτουμε το αντικείμενο στο ArrayList. Τι θα συμβεί αν κάνουμε το αντίθετο, δηλαδή αν προσθέσουμε στο ArrayList το αντικείμενο και στη συνέχεια ενημερώσουμε τις ώρες εργασίας; Ελέγξτε και εξηγήστε τη συμπεριφορά.

13.5.4 Βαθμολογία

Σε ένα τμήμα της Γ’ Γυμνασίου φοιτούν 11 μαθητές. Από αυτούς, οι τρεις είναι αλλοδαποί και οι υπόλοιποι ημεδαποί. Όλοι οι μαθητές παρακολουθούν τα μαθήματα Άλγεβρας, Φυσικής και Γλώσσας. Οι αλλοδαποί παρακολουθούν επιπλέον το μάθημα Εκμάθηση της Ελληνικής γλώσσας. Όλοι οι μαθητές μπορούν να παρακολουθήσουν ή όχι Θρησκευτικά. Το μάθημα έχει επιλέξει ένας αλλοδαπός και έξι ημεδαποί. Όλοι οι μαθητές είναι φυσικά πρόσωπα. Βρισκόμαστε στο τέλος της περιόδου των μαθημάτων, οι μαθητές έχουν βαθμολογηθεί και πρέπει να υπολογίσουμε τον μέσο όρο βαθμολογίας κάθε μαθητή. Φορτώστε σε ένα ArrayList τους 11 μαθητές και εμφανίστε για κάθε έναν τον αριθμό ταυτότητας, το ονοματεπώνυμό του, την αναλυτική βαθμολογία του και τον μέσο όρο του.

Βιβλιογραφία

[1] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition. Boston: Addison-Wesley Professional, 2003.

Κεφάλαιο 14

Σύνοψη
Σε αυτήν την ενότητα καταρχάς εξηγείται η έννοια της διεπαφής στην Java. Στη συνέχεια παρουσιάζονται βασικές προκαθορισμένες διεπαφές και η χρήση τους. Αναλύεται η υλοποίηση διεπαφών οριζόμενων από τον χρήστη. Συζητούνται τα προβλήματα της πολλαπλής κληρονομικότητας και αναλύεται ο μηχανισμός μέσα από τον οποίον υποστηρίζει η Java την πολλαπλή κληρονομικότητα. Εξηγείται η κληρονομικότητα ανάμεσα σε διεπαφές, η ταυτόχρονη κληρονομικότητα από κλάση και διεπαφή, τα στατικά μέλη στις διεπαφές, οι προκαθορισμένες μέθοδοι και οι ανώνυμες κλάσεις.

Προαπαιτούμενη γνώση
Η ενσωμάτωση, η κληρονομικότητα και ο πολυμορφισμός στο αντικειμενοστρεφές μοντέλο με Java, οι αφηρημένες κλάσεις καθώς και οι βασικές προκαθορισμένες κλάσεις, δηλαδή οι String, StringBuilder, ArrayList, LinkedList, Stack, PriorityQueue, HashSet, HashMap, HashTree, LocalDate, Localtime, Period και Duration.

Λέξεις κλειδιά
Διεπαφή (interface), πρόβλημα του διαμαντιού (the diamond problem), πολλαπλή κληρονομικότητα (multiple inheritance), κληρονομικότητα κατάστασης (inheritance of state), κληρονομικότητα υλοποίησης (inheritance of inheritance), κληρονομικότητα τύπου (state inheritance), προκαθορισμένες μέθοδοι (default methods), ανώνυμες κλάσεις (anonymous class).

14 Διεπαφές

Σε πολλές περιπτώσεις, μια ομάδα μηχανικών λογισμικού χρειάζεται να χρησιμοποιήσει λογισμικό που έχει αναπτύξει άλλη ομάδα. Στις περιπτώσεις αυτές χρειάζεται ένας τυπικός τρόπος με τον οποίο η δεύτερη ομάδα πληροφορεί την πρώτη για τα χαρακτηριστικά του λογισμικού της. Αυτός ο τρόπος παρέχεται δια μέσου των διεπαφών (interfaces) [1]. Με άλλα λόγια, όταν ένα λογισμικό Β καλεί ένα λογισμικό Α, τότε το Β χρειάζεται να γνωρίσει τα χαρακτηριστικά του Α. Αυτά ακριβώς τα χαρακτηριστικά εκθέτει το Α προς τους τρίτους δια μέσου των διεπαφών. Πρόκειται δηλαδή για μια συμφωνία μεταξύ των δύο ομάδων μηχανικών λογισμικού ή ακριβέστερα μεταξύ δύο λογισμικών με την οποία καθορίζονται τα χαρακτηριστικά του λογισμικού ώστε η ομάδα-χρήστης να μπορεί να το χρησιμοποιεί χωρίς να γνωρίζει τις εσωτερικές λεπτομέρειές του. Με αυτήν την έννοια, οι διεπαφές αποτελούν έναν ακόμη μηχανισμό αφαιρετικότητας. Αρχικά, οι διεπαφές της Java συνιστούσαν ένα είδος απολύτως αφηρημένης κλάσης, δηλαδή μίας κλάσης στην οποία όλες οι μέθοδοι στιγμιότυπου ήταν αφηρημένες. Από την έκδοση 8 και μετά προστέθηκαν οι προκαθορισμένες μέθοδοι (default methods) που είναι σαφείς. Οι διεπαφές εκτός από αφηρημένες και προκαθορισμένες μεθόδους μπορεί να περιέχουν και στατικές μεθόδους, σταθερές και εμφωλευμένους τύπους. Δεν μπορούν όμως να περιέχουν μεταβλητές στιγμιότυπου. Οι διεπαφές είναι τύποι αναφοράς, ωστόσο δεν μπορεί να δημιουργηθούν στιγμιότυπα διεπαφής για τον ίδιο λόγο που δεν μπορεί να δημιουργηθούν στιγμιότυπα αφηρημένης κλάσης. Μπορούν όμως να δημιουργηθούν στιγμιότυπα των κλάσεων που υλοποιούν (implements) τις διεπαφές. Κάθε κλάση που υλοποιεί μια διεπαφή συνδέεται με τη διεπαφή με τη σχέση “είναι ένα”.

Οι διεπαφές μπορούν να κληρονομούν άλλες διεπαφές. Μάλιστα μια διεπαφή μπορεί να έχει πολλαπλές γονικές διεπαφές. Οι διεπαφές όμως δεν μπορούν να κληρονομήσουν μία κλάση.

Ένα άλλο χαρακτηριστικό των διεπαφών είναι πως για κάθε δημόσια μέθοδο της κλάσης Object, μία αφηρημένη μέθοδος με ίδια ταυτότητα δηλώνεται αυτόματα σε κάθε διεπαφή. Για παράδειγμα, αν δούμε την τεκμηρίωση μιας προκαθορισμένης διεπαφής List θα βρούμε τη μέθοδο boolean equals(Object o). Δημιουργείται έτσι λανθασμένα η εντύπωση πως οι διεπαφές κληρονομούν την κλάση Object. Γενικά οι διεπαφές δεν κληρονομούν από κλάσεις. Ωστόσο, μια διεπαφή μπορεί να κληρονομεί από μία ή περισσότερες διεπαφές.

Πέραν των ανωτέρω, οι διεπαφές αποτελούν τον μηχανισμό μέσα από τον οποίον η Java επιτυγχάνει την πολλαπλή κληρονομικότητα τύπου ενώ ενισχύουν τις δυνατότητες πολυμορφικού κώδικα.

Όλα αυτά θα τα δούμε στη συνέχεια αναλυτικά, αρχίζοντας με την παρουσίαση κάποιων βασικών διεπαφών της βιβλιοθήκης της Java.

14.1 Βασικές διεπαφές

Στις βιβλιοθήκες της Java πέραν των κλάσεων έχουν ορισθεί και πάρα πολλές διεπαφές. Παρουσιάζουμε εδώ συγκεκριμένα τη διεπαφή Iterator και Comparator. Αρχικά μας ενδιαφέρουν κυρίως οι αφηρημένες μέθοδοί τους. Παρακάτω σε αυτό το κεφάλαιο θα συζητήσουμε για τα υπόλοιπα χαρακτηριστικά των διεπαφών.

14.1.1 Iterator

Ο Iterator είναι ίσως η συχνότερα χρησιμοποιούμενη διεπαφή. Μας δίνει τη δυνατότητα να προσπελάσουμε ένα προς ένα τα στοιχεία μιας συλλογής. Δεδομένου ότι οι συλλογές αποτελούν γενικεύσεις, έτσι και ο Iterator αποτελεί μια γενίκευση, π.χ. για να προσπελάσουμε μέσω ενός Iterator τα στοιχεία ενός ArrayList<Person>, χρειαζόμαστε έναν Iterator<Person>. Γενικά, κάθε Iterator αφορά κάποια συλλογή.
Ο Iterator ορίζει δύο αφηρημένες μεθόδους:

```
boolean hasNext()
```

Ελέγχει αν υπάρχει επόμενο στοιχείο στη συλλογή

```
E next()
```

Επιστρέφει το επόμενο διαθέσιμο στοιχείο της συλλογής.

Κάθε μια από τις προκαθορισμένες συλλογές διαθέτει μια μέθοδο που επιστρέφει έναν Iterator με τον οποίο μπορούμε να προσπελάσουμε τα στοιχεία της. Ακολουθεί ένα παράδειγμα χρήσης Iterator με ArrayList.

```
import gr.ihu.cs.lmous.OOJ.k12.v1.x.Person;
import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorDemo {

    public static ArrayList<Person> loadData() {
        ArrayList<Person> rVal = new ArrayList<>();
        rVal.add(new Person("10", "Jim", "Smith", LocalDate.of(1961,
Month.JANUARY, 1)));
        rVal.add(new Person("20", "Adam", "Smith", LocalDate.of(1965,
Month.JANUARY, 1)));
        rVal.add(new Person("1", "John", "Black", LocalDate.of(1971,
Month.JANUARY, 1)));
        rVal.add(new Person("2", "John", "Dalton", LocalDate.of(1965,
Month.JANUARY, 1)));
        rVal.add(new Person("12", "Bill", "Adams", LocalDate.of(1965,
Month.JANUARY, 5)));
        rVal.add(new Person("6", "Averell", "Dalton", LocalDate.of(1972,
Month.JANUARY, 1)));
        rVal.add(new Person("7", "John", "Cyan", LocalDate.of(1980,
Month.JANUARY, 1)));
        return rVal;
    }

    public static void main(String[] args) {
        ArrayList<Person> store = loadData();
        Iterator<Person> it = store.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
```

```
            }
        }
    }
```

Κώδικας 14.1 *Παράδειγμα χρήσης Iterator με ArrayList.*

Στην main του κώδικα 14.1 καλείται καταρχάς η loadData που φορτώνει στο ArrayList, store, τα δεδομένα 7 προσώπων. Η κλάση ArrayList διαθέτει τη μέθοδο iterator που επιστρέφει έναν Iterator για πρόσωπα. Καλώντας την iterator λαμβάνουμε έναν τέτοιον Iterator και τον τοποθετούμε στην αναφορά it. Διαμέσου αυτής της αναφοράς προσπελαύνουμε ένα-ένα τα στοιχεία της ArrayList με την it.next(). Αυτή θα πρέπει να καλείται μόνον εφόσον έχουμε ελέγξει πως υπάρχει διαθέσιμο επόμενο στοιχείο, δηλαδή μόνον εφόσον η it.hasNext() επιστρέψει true. Να σημειωθεί πως σε αυτό το παράδειγμα, η it.next() επιστρέφει ένα αντικείμενο τύπου Person.

Ας δούμε και ένα παράδειγμα που θα βοηθήσει να κατανοήσουμε τις δυνατότητες πολυμορφικού κώδικα που παρέχονται μέσω των διεπαφών.

```
import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class PolymorphismDemo {

      public static List<Person> loadData() {
        List<Person> rVal = new ArrayList<>();
        rVal.add(new Person("10", "Jim", "Smith", LocalDate.of(1961,
Month.JANUARY, 1)));
        rVal.add(new Person("20", "Adam", "Smith", LocalDate.of(1965,
Month.JANUARY, 1)));
        rVal.add(new Person("1", "John", "Black", LocalDate.of(1971,
Month.JANUARY, 1)));
        rVal.add(new Person("2", "John", "Dalton", LocalDate.of(1965,
Month.JANUARY, 1)));
        rVal.add(new Person("12", "Bill", "Adams", LocalDate.of(1965,
Month.JANUARY, 5)));
        rVal.add(new Person("6", "Averell", "Dalton", LocalDate.of(1972,
Month.JANUARY, 1)));
        rVal.add(new Person("7", "John", "Cyan", LocalDate.of(1980,
Month.JANUARY, 1)));
        return rVal;
    }

    public static void main(String[] args) {
        List<Person> store = loadData();
        for (int i=0; i<2; i++) {
            Iterator<Person> it = store.iterator();
            while (it.hasNext()) {
                System.out.println(it.next());
            }
            System.out.println("-------------------------------------");
            store = new LinkedList<>(store);
        }
    }
}
```

Κώδικας 14.2 *Προσπέλαση των στοιχείων μιας ArrayList και μιας LinkedList με τον ίδιο κώδικα*

Στον κώδικα 14.2 η loadData έχει τροποποιηθεί ώστε ο τύπος επιστροφής της δεν είναι ArrayList<Person> όπως στον κώδικα 14.1 αλλά List<Person>. Η List είναι προκαθορισμένη διεπαφή. Τόσο η κλάση ArrayList

όσο και η κλάση LinkedList υλοποιούν τη διεπαφή List. Αυτό σημαίνει πως μια ArrayList είναι μια List και μια LinkedList είναι επίσης μία List. Επομένως, η loadData που η τυπική τιμή επιστροφής της είναι List<Person> μπορεί να επιστρέψει πραγματικό αντικείμενο τύπου ArrayList<Person> ή LinkedList<Person> ή οποιασδήποτε άλλης συλλογής που υλοποιεί την List.

Στην main ορίζουμε επίσης τη μεταβλητή store ως List<Person>. Επομένως στη μεταβλητή αυτή μπορεί να εκχωρηθεί αναφορά σε πραγματικό αντικείμενο οποιασδήποτε κλάσης υλοποιεί την List. Αν μεταβούμε στην τεκμηρίωση της List θα διαπιστώσουμε ότι ανάμεσα σε άλλες μεθόδους ορίζει και μία μέθοδο Iterator<E> iterator(). Κάθε σαφής κλάση που υλοποιεί τη διεπαφή List εγγυάται τη διαθεσιμότητα αυτής της μεθόδου. Με άλλα λόγια σε σχέση με το παράδειγμά μας, τόσο η ArrayList όσο και η LinkedList διαθέτουν μέθοδο Iterator<E> iterator().

Στην main επομένως εκχωρούμε στην store την αναφορά του ArrayList που επέστρεψε η LoadData. Στη συνέχεια, όπως ακριβώς και στον κώδικα 14.1 λαμβάνουμε τον Iterator δια μέσου του οποίου προσπελαύνουμε τα στοιχεία του ArrayList. Όμως, τον κώδικα που προσπελαύνει τα στοιχεία της λίστας τον επαναλαμβάνουμε 2 φορές. Τη δεύτερη φορά έχουμε αλλάξει τη λίστα μας από ArrayList<Person> σε LinkedList<Person>. Ωστόσο ο κώδικας τρέχει σωστά και για τις δύο διαφορετικές δομές, καθώς και οι δύο είναι τύπου List και άρα και οι δύο παρέχουν Iterator. Να σημειωθεί ότι ο Iterator που παρέχεται σε κάθε περίπτωση βασίζεται σε μια κλάση που υλοποιεί τη διεπαφή Iterator. Η ίδια η κλάση όμως είναι διαφορετική για την ArrayList και διαφορετική για την LinkedList.

14.1.2 Comparator

Μερικές φορές έχουμε την ανάγκη να ταξινομήσουμε τα αντικείμενα μιας συλλογής κατά ποικίλους τρόπους. Για παράδειγμα, έστω ότι θέλουμε να εμφανίσουμε τα πρόσωπα της λίστας του κώδικα 14.2 καταρχάς ταξινομημένα κατά id και στη συνέχεια ταξινομημένα κατά επώνυμο και όνομα. Σε αυτές τις περιπτώσεις είναι χρήσιμη η διεπαφή Comparator<T>. Η διεπαφή δηλώνει μια αφηρημένη μέθοδο ως εξής:

```
int compare(T o1, T o2)
```

Συγκρίνει τις δύο παραμέτρους της. Επιστρέφει θετικό ακέραιο αν θεωρεί το o1 μεγαλύτερο από το o2, αρνητικό αν θεωρεί το o1 μικρότερο από το o2 και 0 αν τα θεωρεί ίσα.

Ας δούμε ένα παράδειγμα:

```
import gr.ihu.cs.lmous.OOJ.k12.v1.x.Person;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;

public class ComparatorDemo {

    private static class CompareById implements Comparator<Person> {

        @Override
        public int compare(Person o1, Person o2) {
            int id1 = Integer.decode(o1.getId());
            int id2 = Integer.decode(o2.getId());

            return id1 - id2;

        }
    }

    private static class CompareByName implements Comparator<Person> {

        @Override
        public int compare(Person o1, Person o2) {
            String sName1 = o1.getsName();
            String sName2 = o2.getsName();
```

```
                String fName1 = o1.getfName();
                String fName2 = o2.getfName();
                int rVal = sName1.compareTo(sName2);
                if (rVal == 0) {
                    return fName1.compareTo(fName2);
                }
                return rVal;
            }
        }

        static void prt(Iterator<Person> it) {
            while (it.hasNext()) {
                System.out.println(it.next());
            }
        }

        public static void main(String[] args) {
            List<Person> store = PolymorphismDemo.loadData();
            store.sort(new CompareById());
            prt(store.iterator());
            System.out.println("----------------------");
            store.sort(new CompareByName());
            prt(store.iterator());
        }
    }
```

Κώδικας 14.3 *Χρήση Comparator για ταξινόμηση λίστας*

Στην main φορτώνουμε καταρχάς τα πρόσωπα του κώδικα 14.2 στην store. Στη συνέχεια καλούμε την sort. Η συνάρτηση είναι υπερφορτωμένη. Έχουμε ήδη χρησιμοποιήσει την έκδοση που δεν έχει παραμέτρους. Εδώ χρησιμοποιούμε την έκδοση που απαιτεί ένα Comparator ως παράμετρο. Αρχικά περνάμε ένα αντικείμενο της εσωτερικής κλάσης CompareById. Η CompareById υλοποιεί τη διεπαφή Comparator<Person>, δηλαδή τα αντικείμενά της είναι Comparators για αντικείμενα Person. Επομένως, η CompareById πρέπει να παρέχει υλοποίηση της μοναδικής αφηρημένης μεθόδου της διεπαφής, δηλαδή της compare. Έτσι η sort μπορεί με ασφάλεια να καλέσει από τον Comparator την compare ώστε να μπορέσει να ταξινομήσει τα αντικείμενα Person κατά id.

Στη συνέχεια, η prt λαμβάνει ως παράμετρο τον Iterator της store μέσω του οποίου προσπελαύνει και εμφανίζει ένα προς ένα τα αντικείμενα της store.

Αμέσως μετά ταξινομείται η store με τον CompareByName οπότε η επόμενη κλήση της prt εμφανίζει τα πρόσωπα ταξινομημένα κατά επώνυμο και όνομα. Προσέξτε την εσωτερική κλάση CompareByName. Καταρχάς συγκρίνει δύο πρόσωπα κατά επώνυμο. Αν βρεθούν τα επώνυμα ίδια, τότε συγκρίνει κατά μικρό όνομα.

14.2 Υλοποίηση διεπαφής

Σε αυτήν την ενότητα παρουσιάζουμε ένα παράδειγμα δήλωσης και υλοποίησης μιας διεπαφής. Πιο συγκεκριμένα δηλώνουμε μια διεπαφή που προσομοιώνει τον Iterator της Java αλλά είναι εξειδικευμένη ειδικά για συλλογές του τύπου String. Αυτή η επιλογή γίνεται για καθαρά εκπαιδευτικούς λόγους καθώς μια τέτοια διεπαφή ορθότερα θα ήταν παραμετροποιημένος τύπος. Παράλληλα με τη διεπαφή παρουσιάζεται και μια κλάση που αξιοποιεί τη διεπαφή. Ας ξεκινήσουμε με την παρουσίαση της διεπαφής:

```
public interface MyIterator {

    public boolean hasNext();
    public String next();
}
```

Κώδικας 14.4 *Η διεπαφή MyIterator*

Όπως φαίνεται στον κώδικα 14.4, μια διεπαφή ορίζεται παρόμοια με μία κλάση μόνο που χρησιμοποιείται η λέξη-κλεδί interface αντί της class. Παράλληλα οι μέθοδοι της διεπαφής είναι αφηρημένες.
Ο κώδικας που ακολουθεί παρουσιάζει την εφαρμογή της MyIterator:

```
public class ArrayListOfStrings {

    private String[] store = new String[10];
    private int index = 0;

    private class ArrayListIterator implements MyIterator {

        private int current = 0;

        @Override
        public boolean hasNext() {
            return current < size();
        }

        @Override
        public String next() {
            return get(current++);
        }
    }

    private String[] increment(String[] store) {
        String[] rVal = new String[store.length + 10];
        System.arraycopy(store, 0, rVal, 0, store.length);
        return rVal;
    }

    public void add  (String s) {
        if (index == store.length) {
            store = increment(store);
        }
        store[index++] = s;
    }

    public String remove(int idx) {
        if (idx > index || idx < 0) {
            return null;
        }
        String rVal = store[idx];
        for (int i = idx; i < index; i++) {
            store[i] = store[i + 1];
        }
        index--;
        return rVal;
    }

    public String removeLast() {
        return remove(index - 1);
    }

    public String get(int idx) {
        if (idx >= index || idx < 0) {
            throw new IndexOutOfBoundsException();
        }
        return store[idx];
    }

    public boolean isEmpty() {
```

```
        return index == 0;
    }

    public int size() {
        return index;
    }

    public MyIterator myIterator() {
        return new ArrayListIterator();
    }

    public static void main(String[] args) {
        ArrayListOfStrings list = new ArrayListOfStrings();
        list.add("A");
        list.add("B");
        list.add("C");
        list.add("D");
        MyIterator it = list.myIterator();
        while (it.hasNext()) {
            System.out.print(it.next());
        }
        System.out.println();
    }

}
```

Κώδικας 14.5 *Υλοποίηση MyIterator και ArrayListOfStrings*

Καταρχάς, ας μελετήσουμε την κλάση ArrayListOfStrings. Πρόκειται για μια απλή συλλογή. Ως αποθηκευτικό χώρο χρησιμοποιεί τον πίνακα store που αρχικά δημιουργείται με μήκος 10. Η index δείχνει στην πρώτη θέση του πίνακα που είναι ελεύθερη και μπορεί να αποθηκευτεί μία νέα συμβολοσειρά. Κατά τη δημιουργία αντικειμένου ArrayListOfStrings, η index είναι ίση με 0 που σημαίνει ότι ο πίνακας είναι άδειος. Έτσι η μέθοδος isEmpty επιστρέφει true μόνο εφόσον η index είναι ιση με 0. Η add ελέγχει αν υπάρχει χώρος στον store. Αν όχι καλεί την increment και αυξάνει τη χωρητικότητα του store κατά 10. Η remove διαγράφει το στοιχείο στη θέση idx από τον εσωτερικό πίνακα μετακινώντας όλα τα στοιχεία από τη θέση idx και μετά μία θέση πριν την τρέχουσα θέση τους. Αυτό βέβαια μπορεί να γίνει εφόσον κληθεί με παράμετρο μεγαλύτερη από το 0 και μικρότερη από την index. Ενημερώνει κατάλληλα την index και επιστρέφει το στοιχείο που διαγράφηκε. Η get επιστρέφει το στοιχείο στη θέση idx και η size τον αριθμό των συμβολοσειρών της ArrayListOfString.

Ας πάμε τώρα στην myIterator. Η τυπική τιμή επιστροφής της είναι MyIterator. Έτσι επιστρέφει ένα αντικείμενο της κλάσης ArrayListIterator που υλοποιεί τη διεπαφή MyIterator. Η ArrayListIterator διατηρεί τον δείκτη current που καθορίζει ποιο είναι για το στιγμιότυπο τύπου MyIterator το τρέχον στοιχείο της ArrayListOfString. Κατά τη δημιουργία των αντικειμένων ArrayListIterator, ο δείκτης current αρχικοποιείται στο 0. Έτσι κάθε νέο αντικείμενο ArrayListIterator θεωρεί ως πρώτο στοιχείο αυτό που βρίσκεται στη θέση 0 του store. Η hasNext επιστρέφει true εφόσον ο current είναι μικρότερος από την τιμή που επιστρέφει η size. Η δε next() επιστρέφει το στοιχείο στη θέση current του store και αυξάνει τον current κατά 1.

Η έξοδος της main είναι ABCD.

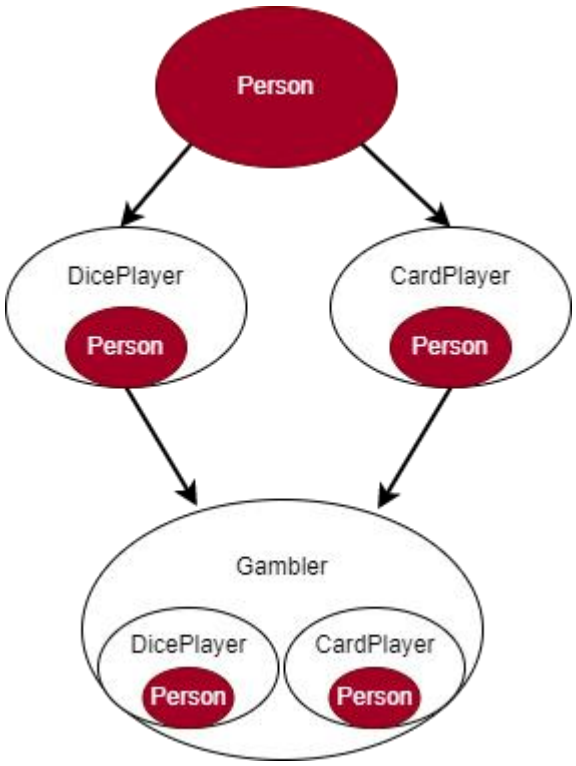
Ας σημειωθεί πως η ArrayListIterator θα μπορούσε να υλοποιεί την προκαθορισμένη διεπαφή Iterator της Java. Χρησιμοποιήθηκε εδώ η MyIterator ως μια ευκαιρία για να επιδειχθεί πως δηλώνουμε μια δική μας διεπαφή.

14.3 Πολλαπλή κληρονομικότητα

Ας υποθέσουμε πως έχουμε μια κλάση που αναπαριστά παίκτες ζαριών, έστω την DicePlayer, και μία κλάση που αναπαριστά χαρτοπαίκτες, έστω την CardPlayer. Έστω ότι χρειαζόμαστε και μία κλάση τα αντικείμενα της οποίας είναι ταυτοχρόνως και χαρτοπαίκτες και παίκτες ζαριού, έστω Gambler. Θα ήταν ίσως βολική η δυνατότητα η κλάση Gambler να κληρονομεί και την CardPlayer και την DicePlayer, δηλαδή η Gambler να έχει 2 γονικές κλάσεις. Εύκολα μπορούμε να σκεφτούμε παραδείγματα που ενδεχομένως θα ήταν χρήσιμο μια

κλάση να έχει περισσότερες ακόμη και από 2 γονικές. Η σχέση κατά την οποία μία κλάση διαθέτει περισσότερες από μία γονικές είναι γνωστή ως πολλαπλή κληρονομικότητα (multiple inheritance).

Η πολλαπλή κληρονομικότητα μεταξύ κλάσεων χαρακτηρίζεται από το πρόβλημα του διαμαντιού (the diamond problem). Ανακαλέστε από την ενότητα 13.1 πως κάθε αντικείμενο παράγωγης κλάσης περιλαμβάνει ένα αντικείμενο της γονικής.



Σχήμα 14.1 *Το πρόβλημα του διαμαντιού*

Προσέξτε τώρα στο σχήμα 14.1 τι συμβαίνει αν η κλάση DicePlayer και η CardPlayer είναι παράγωγες της Person. Ένα αντικείμενο Person εσωκλείεται σε κάθε στιγμιότυπο της DicePlayer και της CardPlayer. Έτσι κάθε αντικείμενο της Gambler καταλήγει να περιλαμβάνει δύο αντικείμενα της Person. Αυτό βέβαια είναι μη αποδεκτό, καθώς αυξάνει ανώφελα τον όγκο των αντικειμένων Gambler και θέτει το ακόλουθο ερώτημα: Τι θα πρέπει να συμβεί αν αντικείμενο της Gambler καλέσει μια μέθοδο που αρχικά ορίζεται στην Person, π.χ. την getfName. Προσέξτε πως η getfName κληρονομείται και στην DicePlayer και στην CardPlayer. Επίσης λάβετε υπόψη πως η μέθοδος μπορεί να επανοριστεί ή όχι τόσο στην DicePlayer όσο και στην CardPlayer. Ποια getfName θα κληρονομήσει η Gambler; Επιπλέον, οι μεταβλητές στιγμιότυπου της Parent έχουν δημιουργηθεί και λάβει τιμές τόσο κατά τη δημιουργία του αντικειμένου DicePlayer όσο και κατά τη δημιουργία του CardPlayer. Ποιες μεταβλητές θα κληρονομήσει η Gambler; Αν υποθέσουμε πως η Gambler κληρονομεί τις μεταβλητές της Parent που δημιουργήθηκαν κατά την κατασκευή του αντικειμένου DicePlayer, οι μέθοδοι που κληρονομήθηκαν από την CardPlayer θα μπορούν να επενεργούν σε αυτές τις μεταβλητές;

Οι διάφορες σύγχρονες γλώσσες έχουν δώσει διαφορετική απάντηση στο πρόβλημα του διαμαντιού. Ας αποσαφηνίσουμε όμως κάποιες έννοιες σχετικά με την κληρονομικότητα ώστε να διευκολυνθούμε για να κατανοήσουμε τη λύση της Java.

Η κληρονομικότητα μπορεί να διακριθεί σε τρεις τύπους [2], την κληρονομικότητα κατάστασης (state inheritance), την κληρονομικότητα υλοποίησης (implementation inheritance) και την κληρονομικότητα τύπου (type inheritance). Η κληρονομικότητα κατάστασης αφορά τις μεταβλητές που κληρονομούνται από τη γονική, η κληρονομικότητα υλοποίησης τις υλοποιήσεις των μεθόδων που κληρονομούνται από τη γονική και τέλος η κληρονομικότητα τύπων αφορά τους τύπους που κληρονομούνται από τη γονική. Η κληρονομικότητα μεταξύ των κλάσεων περιλαμβάνει και τους τρεις τύπους κληρονομικότητας. Το προβλήματος του διαμαντιού όμως προκύπτει αποκλειστικά σε σχέση με την κληρονομικότητα κατάστασης και υλοποίησης. Επιπλέον, η κληρονομικότητα κατάστασης και υλοποίησης δεν είναι αναγκαίες προϋποθέσεις για την ανάπτυξη

πολυμορφικού κώδικα. Αντίθετα η κληρονομικότητα τύπου είναι αναγκαία προϋπόθεση για την υποστήριξη πολυμορφικού κώδικα. Η Java δεν υποστηρίζει πολλαπλή κληρονομικότητα κατάστασης και υλοποίησης. Έτσι αντιμετωπίζει το πρόβλημα του διαμαντιού. Υποστηρίζει όμως πολλαπλή κληρονομικότητα τύπου δια μέσου των διεπαφών. Πράγματι οι διεπαφές διαθέτουν μόνο αφηρημένες μεθόδους και δεν επιτρέπουν τη δήλωση μεταβλητών στιγμιότυπου. Μία κλάση της Java μπορεί να κληρονομήσει μόνο από μία κλάση αλλά μπορεί να υλοποιήσει απεριόριστο αριθμό διεπαφών. Τα αντικείμενα της κλάσης συνδέονται με τη σχέση "είναι ένα" με κάθε διεπαφή που η κλάση υλοποιεί.

Ας δούμε τώρα πώς μπορούμε να αναπτύξουμε την κλάση Gambler στο πλαίσιο της Java έτσι ώστε κάθε αντικείμενο Gambler να είναι ταυτοχρόνως Person και DicePlayer και CardPlayer. Θα χρειαστούμε καταρχάς μια βοηθητική κλάση που αναπαριστά το ζάρι και δύο διεπαφές.

```
import gr.ihu.cs.lmous.OOJ.k11.v1.x.Lib;
import java.util.Arrays;

public class Dice {
    private final double[] distribution;

    public Dice(double[] distribution) {

        if (distribution.length != 6
                || !Lib.approximateEquals(Lib.sum(distribution), 1)) {
            throw new RuntimeException();
        }
        this.distribution = distribution;
    }

    public Dice() {
        distribution=new double[6];
        Arrays.fill(distribution, 1/6d);
    }

    public int roll() {
        double value = Lib.gen.nextDouble();
        double sum = 0;

        for (int i = 0; i < distribution.length; i++) {
            sum += distribution[i];
            if (value <= sum) {
                return i + 1;
            }
        }
        return 0;
    }
}

public interface DicePlayerI {

    public int roll(Dice dice);
}

import java.util.ArrayList;

public interface CardPlayerI {

    public Card pickTop(Deck deck);

    public ArrayList<Card> openCards();
}
```

Κώδικας 14.6 *Η κλάση Dice και οι διεπαφές DicePlayerI και CardPlayerI*

Σημειώστε πως η κλάση Dice και οι διεπαφές DicePlayerI και CardPlayerI πρέπει να τοποθετηθούν σε διαφορετικά αρχεία. Έχοντας στη διάθεσή μας τις δύο αυτές διεπαφές και την κλάση Dice, μπορούμε να ορίσουμε την κλάση Gambler που παρουσιάζεται στον κώδικα 14.7. Σημειώστε πως εδώ η κατανομή των πιθανοτήτων για το αποτέλεσμα του ζαριού δεν είναι πλέον χαρακτηριστικό του παίκτη αλλά του ζαριού που χρησιμοποιεί:

```
import java.time.LocalDate;
import java.util.ArrayList;

public class Gambler extends Person implements CardPlayerI, DicePlayerI {

    ArrayList<Card> cardsInHand = new ArrayList<>();

    public Gambler(String id, String fName, String sName, LocalDate
birthday) {
        super(id, fName, sName, birthday);
    }

    @Override
    public Card pickTop(Deck deck) {
        Card card = deck.pickCard();
        cardsInHand.add(card);
        return card;
    }

    @Override
    public int roll(Dice dice) {
        return dice.roll();
    }

    @Override
    public ArrayList<Card> openCards() {
        ArrayList<Card> rVal = new ArrayList<>(cardsInHand);
        cardsInHand.clear();
        return rVal;
    }
}
```

Κώδικας 14.7 *Η κλάση Gambler*

Η κλάση Gambler στον κώδικα 14.7 κληρονομεί από την Person (κώδικας 12.19) άρα διαθέτει τους τύπους, τις υλοποιήσεις και τις μεταβλητές της Person. Κάθε αντικείμενο Gambler επομένως είναι ένα Person. Επιπλέον υλοποιεί τις διεπαφές CardPlayerI και DicePlayerI. Αυτό σημαίνει πως για να είναι σαφής και όχι αφηρημένη, η Gambler πρέπει να παρέχει υλοποιήσεις για τις αφηρημένες μεθόδους των διεπαφών. Επομένως, κάθε αντικείμενο Gambler μπορεί να επιτελέσει τις λειτουργίες που ορίζονται στις CardPlayerI και DicePlayerI. Συνεπώς, κάθε αντικείμενο Gambler θεωρείται πως είναι ένας παίκτης ζαριών και ένας χαρτοπαίκτης. Ο κώδικας 14.8 επιδεικνύει τη χρήση της Gambler.

```
import java.time.LocalDate;
import java.time.Month;

public class GamblerDemo {

    public static void main(String[] args) {
        Deck deck = new Deck();
        double[] fairDist = new double[]{1d / 6, 1d / 6, 1d / 6, 1d / 6, 1d
/ 6, 1d / 6,};
        Dice dice = new Dice(fairDist);
```

```
        Gambler jim = new Gambler("01", "Jim", "Black", LocalDate.of(1976,
Month.JULY, 5));
        Gambler jack = new Gambler("02", "Jack", "Green", LocalDate.of(1981,
Month.APRIL, 12));
        deck.shuffle();
        for (int i = 0; i < 4; i++) {
            jim.pickTop(deck);
            jack.pickTop(deck);
        }
        System.out.println("Οι κάρτες του " + jim.getsName() + " " +
jim.openCards());
        System.out.println("Οι κάρτες του " + jack.getsName() + " " +
jack.openCards());
        System.out.println("Ζαριά του " + jim.getsName() + " " +
jim.roll(dice));
        System.out.println("Ζαριά του " + jack.getsName() + " " +
jack.roll(dice));
        CardPlayerI jimCP = jim;
        DicePlayerI jackDP = jack;
        Person jimP = jim;
    }
}
```

Κώδικας 14.8 *Κάθε αντικείμενο της κλάσης Gambler είναι Person και DicePlayerI και CardPlayerI*

Εδώ ορίζουμε δύο αντικείμενα τύπου Gambler. Στη συνέχεια ανακατεύουμε μια τράπουλα και τους δίνουμε να τραβήξουν από τέσσερις κάρτες ο κάθε ένας. Μετά ανοίγουν τις κάρτες τους τις οποίες εμφανίζουμε συνοδευόμενες από το επώνυμο του αντίστοιχου παίκτη. Επομένως, μέχρι εδώ είναι σαφές πως κάθε αντικείμενο Gambler είναι ένα πρόσωπο και ένας χαρτοπαίκτης ταυτοχρόνως. Στο επόμενο βήμα τους βάζουμε να ρίξουν από μία ζαριά. Ας σημειωθεί πως η κλήση Dice dice = new Dice(fairDist) θα μπορούσε να αντικατασταθεί με την κλήση Dice dice=new Dice() με ίδιο αποτέλεσμα.

Οι τελευταίες τρεις γραμμές της main έχουν ως σκοπό να επιδείξουν πως αυτά τα αντικείμενα μπορούμε να τα δούμε μέσα στον κώδικά μας αποκλειστικά είτε σαν Person είτε σαν DicePlayer είτε σαν CardPlayer.

14.4 Άλλα χαρακτηριστικά

14.4.1 Κληρονομικότητα διεπαφών

Οι διεπαφές δεν μπορούν να κληρονομήσουν από κλάσεις. Ο κώδικας

```
interface Student extends Person {}
```

παράγει λάθος μεταγλώττισης.

Οι διεπαφές μπορούν να κληρονομήσουν από απεριόριστο αριθμό διεπαφών. Ο κώδικας

```
public interface Gambler extends CardPlayerI, DicePlayerI {}
```

είναι ορθός.

14.4.2 Η κλάση νικάει

Στην περίπτωση που μία κλάση κληρονομεί τον ίδιο τύπο από κλάση και διεπαφή, τότε κληρονομείται η υλοποίηση της κλάσης και ο κοινός τύπος δεν είναι υποχρεωτικό να επανοριστεί στην παράγωγη κλάση.

```
import java.time.LocalDate;
import java.time.Month;
```

```
interface Named {
    public String getfName();
    public String getsName();
}

public class Inherited extends Person implements Named {

    public Inherited(String id, String fName, String sName, LocalDate
birthday) {
        super(id, fName, sName, birthday);
    }

    public static void main(String[] args) {
        Inherited i=new Inherited("1", "George", "Best",
LocalDate.of(1950,Month.MARCH, 1));
        System.out.println(i.getfName());
    }
}
```

Κώδικας 14.9 *Η κλάση Inherited κληρονομεί ίδιους τύπους από τη διεπαφή Named και την κλάση Person*

Η κλάση Inherited είναι παράγωγη της Person. Επομένως κληρονομεί τις μεθόδους της Person ανάμεσα στις οποίες συμπεριλαμβάνονται οι getfName() και getsName(). Παράλληλα, η κλάση υλοποιεί τη διεπαφή Named που ορίζει τις ίδιες μεθόδους. Σε αυτήν την περίπτωση ισχύει ο κανόνας που λέει πως η κλάση νικάει (calss wins), δηλαδή η Inherited κληρονομεί τις υλοποιήσεις της Person. Επομένως, έχει τη δυνατότητα να τις επανορίσει χωρίς όμως να είναι υποχρεωτικό.

14.4.3 Στατικά μέλη

Τα στατικά μέλη δεν κληρονομούνται. Τα στατικά μέλη μπορούν να οριστούν στις διεπαφές με τον ίδιο τρόπο που ορίζονται και στις κλάσεις. Η προσπέλαση σε αυτά γίνεται με προσδιοριστή ονόματος το όνομα της διεπαφής ακολουθούμενο από τον τελεστή τελεία και το όνομα του μέλους.

```
import java.util.Arrays;

public class AllaXaraktiristika {

    public interface Gambler extends CardPlayerI, DicePlayerI {
    }

    public interface Distribution {

        public double[] getDistribution();

        public static Double[] fairDistribution() {
            return new Double[]{1d / 6, 1d / 6, 1d / 6, 1d / 6, 1d / 6, 1d
/ 6};
        }
    }

    public static void main(String[] args) {
        Double[] d = Distribution.fairDistribution();
        System.out.println(Arrays.asList(d));
    }
}
```

Κώδικας 14.10 *Στατικά μέλη στις διεπαφές*

Όπως φαίνεται στον κώδικα 14.10, η διεπαφή Distribution ορίζει το αφηρημένο μέλος getDistribution και το στατικό μέλος fairDistribution. Εφόσον η fairDistribution είναι μία για όλες τις υλοποιήσεις της διεπαφής, λογικό είναι να ορισθεί ως στατική. Η fairDistribution μπορεί να κληθεί ως Distribution. fairDistribution().

14.4.4 Προκαθορισμένες μέθοδοι

Οι προκαθορισμένες μέθοδοι (default methods) στις διεπαφές είναι αποτέλεσμα της πολιτικής της Java να διατηρεί συμβατότητα με όλες τις προηγούμενες εκδόσεις της. Θα το εξηγήσουμε αυτό με το παράδειγμα της διεπαφής Iterator. Αρχικά, η διεπαφή όριζε δύο αφηρημένες μεθόδους, την hasNext() και την next(). Κάποια στιγμή έγινε αντιληπτό πως η πρόσθεση μιας μεθόδου remove() που διαγράφει από την υποκείμενη συλλογή το τρέχον στοιχείο του Iterator θα ήταν χρήσιμη σε πολλές περιπτώσεις. Έως ότου να προστεθεί όμως η remove είχε αναπτυχθεί παγκοσμίως σημαντικός όγκος κώδικα που χρησιμοποιούσε τον Iterator υλοποιώντας μόνο την hasNext() και την next(). Εάν λοιπόν προστίθετο στην Iterator η remove ως αφηρημένη μέθοδος, ο μεταγλωττιστής θα απαιτούσε την υλοποίησή της με αποτέλεσμα ο προϋπάρχων κώδικας να μην μεταγλωττίζεται. Ωστόσο ο προϋπάρχων κώδικας δεν αξιοποιούσε την remove, καθώς όταν αναπτύχθηκε δεν υπήρχε. Προκειμένου να αντιμετωπιστεί αυτό το πρόβλημα εισήχθησαν στην έκδοση 8 της Java οι προκαθορισμένες μέθοδοι. Οι προκαθορισμένες μέθοδοι παρέχουν υλοποίηση έτσι ώστε οι κώδικες-πελάτες να περνούν μεταγλώττιση χωρίς να απαιτείται να τις υλοποιήσουν, ενώ παράλληλα νέοι κώδικες που τις χρειάζονται μπορούν να τις επανορίσουν.

Σημειώστε πως η λειτουργικότητα των προκαθορισμένων μεθόδων δεν επαρκεί για να χρησιμοποιηθούν παρά μόνο αν επανοριστούν. Πιο συγκεκριμένα, οι προκαθορισμένες μέθοδοι, εφόσον δεν επανοριστούν, απλά παράγουν μια εξαίρεση που πληροφορεί πως η μέθοδος δεν έχει οριστεί.

Τέλος να σημειωθεί πως ο κανόνας η κλάση νικάει ισχύει και για τις προκαθορισμένες μεθόδους.

14.4.5 Ανώνυμες κλάσεις

Μερικές φορές χρειαζόμαστε μια κλάση μόνο μία φορά μέσα στο πρόγραμμά μας. Σε αυτές τις περιπτώσεις, οι διεπαφές μας δίνουν τη δυνατότητα αξιοποίησης ανώνυμων (anonymous) κλάσεων. Αν υποθέσουμε πως χρειαζόμαστε ένα μόνο αντικείμενο τύπου DicePlayerI, τότε μπορούμε να το δημιουργήσουμε όπως δείχνει ο κώδικας 14.11, δηλαδή με τον τελεστή new ακολουθούμενο από τη διεπαφή DicePlayerI και ένα μπλοκ στο οποίο επανορίζουμε τις μεθόδους της διεπαφής και προσθέτουμε οποιαδήποτε άλλα μέλη χρειαζόμαστε. Με αυτόν τον τρόπο έχουμε ένα αντικείμενο με τα χαρακτηριστικά που χρειαζόμαστε, εκεί ακριβώς που το χρειαζόμαστε. Ωστόσο, σε γενικές γραμμές δεν συνίσταται η χρήση μεγάλων και πολύπλοκων ανώνυμων κλάσεων.

```
public class AnonymousDemo {

    public static void main(String[] args) {
        Dice dice = new Dice();
        DicePlayerI jim = new DicePlayerI() {
            @Override
            public int roll(Dice dice) {
                return dice.roll();
            }
        };
        for (int i = 0; i < 50; i++) {
            System.out.println(jim.roll(dice));
        }
    }
}
```

Κώδικας 14.11 *Επίδειξη της ανώνυμης κλάσης*

14.5 Λυμένες Ασκήσεις

14.5.1. Καζίνο

Σε ένα καζίνο υπάρχει η δυνατότητα να παίξει κανείς διαφορετικά παιχνίδια με τράπουλα. Κάποια παιχνίδια απαιτούν και τις 52 κάρτες της τράπουλας και κάποια παίζονται με τράπουλα που περιέχει τις κάρτες από 7 έως τον άσο, δηλαδή με 32 κάρτες. Έτσι το καζίνο χρειάζεται 2 τύπους τράπουλας.
Δημιουργήστε κατάλληλη διεπαφή τράπουλας και δύο κλάσεις που υλοποιούν τη διεπαφή. Η μία κλάση θα αναπαριστά τράπουλα 52 καρτών και η άλλη τράπουλα 32 καρτών. Σε κάθε μία από αυτές τις κλάσεις, υλοποιήστε τον αποθηκευτικό χώρο ως πίνακα από κάρτες. Αναπτύξτε εφαρμογή που αποθηκεύει μια τράπουλα 32 καρτών και μία 52 καρτών σε κατάλληλο πίνακα. Στη συνέχεια για κάθε τράπουλα στον πίνακα, ανακατέψτε την, σηκώστε μια κάρτα, εμφανίστε την κάρτα και στη συνέχεια εμφανίστε την υπόλοιπη τράπουλα. Ελέγξτε την ορθότητα της εξόδου.

Λύση
Καταρχάς, παρουσιάζεται η διεπαφή:

```
public interface Deck {
    public void shuffle();
    public boolean empty();
    public Card pickTop();
}
```

Κώδικας 14.12 *Η διεπαφή Deck*

Οι μέθοδοι shuffle και pickTop απαιτούνται άμεσα από την εκφώνηση. Η μέθοδος empty που ελέγχει αν υπάρχει ή όχι στην τράπουλα κάρτα είναι γενικότερα χρήσιμη. Στη συνέχεια δίνεται η κλάση Deck52 που αναπαριστά τράπουλες των 52 καρτών.

```
import java.util.Random;

public class Deck52 implements Deck {

    final Card[] deck;
    private int cardsInDeck;
    private static final int DECKSIZE = 52;

    public Deck52() {
        deck = new Card[DECKSIZE];
        cardsInDeck = 0;
        for (CardRank r : CardRank.values()) {
            for (Suit s : Suit.values()) {
                deck[cardsInDeck++] = new Card(r, s);
            }
        }
    }

    @Override
    public void shuffle() {
        Random r = new Random();

        for (int i = 0; i < deck.length; i++) {
            int rIdx = r.nextInt(deck.length);
            Card tmp = deck[i];
            deck[i] = deck[rIdx];
            deck[rIdx] = tmp;
        }
    }

    @Override
    public boolean empty() {
        return cardsInDeck == 0;
    }
```

```
    @Override
    public Card pickTop() {
        cardsInDeck--;
        return deck[DECKSIZE - cardsInDeck - 1];
    }

    @Override
    public String toString() {
        if (cardsInDeck == 0) {
            return "Empty Deck";
        }
        StringBuilder rVal = new StringBuilder();
        for (int i = DECKSIZE - cardsInDeck; i < DECKSIZE; i++) {
            if (i > DECKSIZE - cardsInDeck && i % 4 == 0) {
                rVal.append("\n");
            }
            rVal.append(deck[i] + " ");
        }
        return rVal.toString();
    }
}
```

Κώδικας 14.13 *Η κλάση Deck52*

Παρόμοια είναι και η κλάση Deck32 που αναπαριστά τράπουλες των 32 καρτών.

```
import java.util.Random;

public class Deck32 implements Deck {

    private final Card[] deck;
    private int cardsInDeck;
    private static final int DECKSIZE = 32;

    public Deck32() {
        deck = new Card[DECKSIZE];
        cardsInDeck = 0;
        for (int i = 5; i < 13; i++) {
            CardRank r = CardRank.values()[i];
            for (Suit s : Suit.values()) {
                deck[cardsInDeck++] = new Card(r, s);
            }
        }
    }

    @Override
    public void shuffle() {
        Random r = new Random();

        for (int i = 0; i < deck.length; i++) {
            int rIdx = r.nextInt(deck.length);
            Card tmp = deck[i];
            deck[i] = deck[rIdx];
            deck[rIdx] = tmp;
        }
    }

    @Override
    public boolean empty() {
        return cardsInDeck == 0;
    }
```

```
    @Override
    public Card pickTop() {
        cardsInDeck--;
        return deck[DECKSIZE - cardsInDeck - 1];
    }

    @Override
    public String toString() {
        if (cardsInDeck == 0) {
            return "Empty Deck";
        }
        StringBuilder rVal = new StringBuilder();
        for (int i = DECKSIZE - cardsInDeck; i < DECKSIZE; i++) {
            if (i > DECKSIZE - cardsInDeck && i % 4 == 0) {
                rVal.append("\n");
            }
            rVal.append(deck[i] + " ");
        }
        return rVal.toString();
    }
}
```

Κώδικας 14.14 *Η κλάση Deck32*

Στον κώδικα 14.15 δίνεται και η εφαρμογή που ζητείται από την εκφώνηση.

```
public class App {
    public static void main(String[] args) {
        Deck[] decks=new Deck[2];
        decks[0]=new Deck32();
        decks[1]=new Deck52();
        for (Deck deck:decks) {
            deck.shuffle();
            System.out.println(deck.pickTop());
            System.out.println("------------------------------");
            System.out.println(deck);
            System.out.println("------------------------------");
        }
    }
}
```

Κώδικας 14.15 *Η ζητούμενη εφαρμογή*

14.5.2 Τζόγος στου John

Στη συγκέντρωση που διοργάνωσε ο John Beker μαζεύτηκαν 14 πρόσωπα. Σε κάποιους από τους καλεσμένους αρέσει να παίζουν ζάρια. Ο John έχει 3 ζάρια. Κάποια ζάρια είναι κανονικά και φέρνουν κάθε αποτέλεσμα από 1 έως 6 με πιθανότητα 1/6, κάποια άλλα όμως δεν φέρνουν ποτέ 5 και φέρνουν 6 με πιθανότητα 1/3. Τέλος, υπάρχουν και ζάρια που δεν φέρνουν 4 ποτέ αλλά φέρνουν 5 με πιθανότητα 1/3.

Επίσης, κάποιοι από τους καλεσμένους παίζουν χαρτιά.

Στους καλεσμένους του John συμπεριλαμβάνονται κάποιοι που παίζουν αποκλειστικά ζάρια, κάποιοι που παίζουν αποκλειστικά χαρτιά αλλά και κάποιοι που παίζουν και χαρτιά και ζάρια.

Σε μια παρτίδα ζαριών συμμετέχουν 2 παίκτες. Ρίχνουν εναλλάξ από 10 ζαριές. Κάθε φορά που οι 2 παίκτες ρίχνουν το ζάρι είτε ο ένας φέρνει μεγαλύτερο αποτέλεσμα από τον άλλον είτε φέρνουν το ίδιο αποτέλεσμα. Στην πρώτη περίπτωση ενημερώνεται αθροιστικά το σκορ του παίκτη που έφερε το μεγαλύτερο αποτέλεσμα. Στη δεύτερη περίπτωση δεν ενημερώνεται κανένα σκορ.

Σε μια παρτίδα χαρτιών συμμετέχουν όλοι οι διαθέσιμοι (χαρτο)παίκτες. Στην αρχή της παρτίδας ανακατεύουν την τράπουλα και στη συνέχεια τραβάνε από μια κάρτα εναλλάξ έως ότου τελειώσουν οι κάρτες. Στο τέλος της παρτίδας, ο κάθε παίκτης παρουσιάζει τα χαρτιά που έχει μαζέψει.

Ζητείται να αναπτυχθεί εφαρμογή που:

1. Βάζει κάθε πρόσωπο στη συγκέντρωση του John να συναντηθεί με κάθε άλλο πρόσωπο. Αν κατά τη συνάντηση είναι και οι 2 παίκτες ζαριού, τότε παίζουν μια παρτίδα ζάρια. Αν όμως κατά τη συνάντηση κάποιος είναι χαρτοπαίκτης, τότε εγγράφεται στην ομάδα των χαρτοπαικτών.
2. Όλοι οι εγγεγραμμένοι χαρτοπαίκτες συμμετέχουν σε μια παρτίδα χαρτιών.
3. Εμφανίζει το ονοματεπώνυμο και σκορ για κάθε παίκτη ζαριού κατά φθίνουσα σειρά των σκορ.
4. Νικητής στα χαρτιά ανακηρύσσεται αυτός που έχει μαζέψει τους περισσότερους άσους. Αν 2 ή περισσότεροι παίκτες έχουν τον ίδιο μέγιστο αριθμό άσων, τότε ανακηρύσσονται νικητές.

Λύση
Καταρχάς, θα πρέπει τώρα να σχεδιάσουμε μια κλάση για τα πρόσωπα που είναι παίκτες ζαριού, μια για όσους είναι χαρτοπαίκτες και μία για όσους παίζουν και χαρτιά και ζάρια. Και οι τρεις κλάσεις είναι παράγωγες της κλάσης Person. Επιπλέον, η κλάση των χαρτοπαικτών υλοποιεί τη διεπαφή CardPlayerI, η κλάση των παικτών ζαριού υλοποιεί τη διεπαφή DicePlayerI και η κλάση των προσώπων που είναι και χαρτοπαίκτες και παίκτες ζαριού υλοποιεί και την DicePlayerI και την CardPlayerI. Την τελευταία κλάση, την έχουμε υλοποιήσει ήδη (κώδικας 14.7).

```
import java.time.LocalDate;
import java.util.ArrayList;

public class CardPlayer extends Person implements CardPlayerI {

    ArrayList<Card> cardsInHand = new ArrayList<>();

    public CardPlayer(String id, String fName, String sName, LocalDate
birthday) {
        super(id, fName, sName, birthday);
    }

    @Override
    public Card pickTop(Deck deck) {
        Card card = deck.pickCard();
        cardsInHand.add(card);
        return card;
    }

    @Override
    public ArrayList<Card> openCards() {
        ArrayList<Card> rVal = new ArrayList<>(cardsInHand);
        cardsInHand.clear();
        return rVal;
    }
}
```

Κώδικας 14.16 *Η κλάση CardPlayer*

Η κλάση CardPlayer διαθέτει ArrayList<Card> στο οποίο αποθηκεύεται κάθε κάρτα που λαμβάνει ο παίκτης από τράπουλα. Τα αντικείμενα δημιουργούνται με κλήση στον δημιουργό της γονικής κλάσης Person. Η pickTop λαμβάνει μια κάρτα από την τράπουλα που της δίνεται παραμετρικά, η openCards αντιγράφει τις κάρτες που έχει ο παίκτης στα χέρια του σε ένα άλλο ArrayList, στη συνέχεια, διαγράφει τις κάρτες στα χέρια του παίκτη και επιστρέφει το αντίγραφο.

```
import java.time.LocalDate;

public class DicePlayer extends Person implements DicePlayerI {

    public DicePlayer(String id, String fName, String sName, LocalDate
birthday) {
        super(id, fName, sName, birthday);
```

```
    }

    @Override
    public int roll(Dice dice) {
        return dice.roll();
    }
}
```

Κώδικας 14.17 *Η κλάση DicePlayer*

Ο κώδικας 14.73 παρουσιάζει την κλάση παικτών ζαριού. Η κλάση Dice έχει δοθεί ήδη στον κώδικα 14.6. Η κλάση Person είναι η έκδοση που δίνεται στον κώδικα 12.19. Με τις κλάσεις αυτές στη διάθεσή μας μπορούμε να προχωρήσουμε στην ανάπτυξη της εφαρμογής.

```
import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Random;
import java.util.Set;

public class BeckersMeeting {

    private static final Map<DicePlayerI, Integer> diceScores = new
LinkedHashMap<>();
    private static final Dice fair = new Dice();
    private static final Dice two6 = new Dice(new double[]{1. / 6, 1. / 6,
1. / 6, 1. / 6, 0, 1. / 3,});
    private static final Dice two5 = new Dice(new double[]{1. / 6, 1. / 6,
1. / 6, 0, 1. / 3, 1. / 6,});
    private static final Dice[] dices = new Dice[]{fair, two6, two5};
    private final static Random gen = new Random();

    static Dice pickDice() {
        return dices[gen.nextInt(3)];
    }

    private static void diceGame(DicePlayerI p1, DicePlayerI p2) {
        int score1 = 0;
        int score2 = 0;
        for (int i = 0; i < 10; i++) {
            int s1 = p1.roll(pickDice());
            int s2 = p2.roll(pickDice());
            if (s1 > s2) {
                score1 += s1;
            } else if (s2 > s1) {
                score2 += s2;
            }
        }

        Integer score = diceScores.get(p1);
        if (score == null) {
            score = 0;
        }
```

```
        diceScores.put(p1, score1 + score);

        score = diceScores.get(p2);
        if (score == null) {
            score = 0;
        }
        diceScores.put(p1, score2 + score);
    }

    private static void cardGame(Set<CardPlayerI> cardPlayers) {
        Deck deck = new Deck();
        deck.shuffle();
        while (!deck.empty()) {
            for (CardPlayerI cp : cardPlayers) {
                cp.pickTop(deck);
                if (deck.empty()) {
                    break;
                }
            }
        }
    }

    private static void meet(Person p1, Person p2, Set<CardPlayerI>
cardPlayers) {
        if (p1 instanceof DicePlayerI && p2 instanceof DicePlayerI) {
            diceGame((DicePlayerI) p1, (DicePlayerI) p2);
        }

        if (p1 instanceof CardPlayerI) {
            cardPlayers.add((CardPlayerI) p1);
        }

        if (p2 instanceof CardPlayerI) {
            cardPlayers.add((CardPlayerI) p2);
        }
    }

    static int cntAces(ArrayList<Card> cards) {
        int rVal = 0;
        for (Card card : cards) {
            if (card.getCardRank() == CardRank.ACE) {
                rVal++;
            }
        }
        return rVal;
    }

    public static void main(String[] args) {
        Person[] persons = new Person[14];
        persons[0] = new DicePlayer("100", "Jack", "Antie",
LocalDate.of(2001, Month.MARCH, 1));
        persons[1] = new DicePlayer("101", "John", "Travolta",
LocalDate.of(1961, Month.AUGUST, 3));
        persons[2] = new DicePlayer("102", "Bill", "Allen",
LocalDate.of(1980, Month.FEBRUARY, 4));
        persons[3] = new DicePlayer("103", "Jim", "Morison",
LocalDate.of(2001, Month.JULY, 5));
        persons[4] = new DicePlayer("104", "Peter", "MacMillan",
LocalDate.of(2002, Month.MARCH, 1));
        persons[5] = new DicePlayer("105", "George", "Best",
LocalDate.of(1970, Month.MAY, 1));
```

```
        persons[6] = new DicePlayer("106", "George", "Blue",
LocalDate.of(1980, Month.MAY, 1));
        persons[7] = new DicePlayer("107", "Beny", "Hill",
LocalDate.of(1977, Month.JULY, 21));
        persons[8] = new CardPlayer("108", "John", "Beker",
LocalDate.of(1977, Month.JULY, 23));
        persons[9] = new CardPlayer("109", "Alison", "Stair",
LocalDate.of(1978, Month.APRIL, 22));
        persons[10] = new CardPlayer("110", "Erik", "Erikson",
LocalDate.of(1979, Month.JANUARY, 13));

        persons[11] = new Gambler("111", "Erik", "Johnson",
LocalDate.of(1989, Month.JANUARY, 13));
        persons[12] = new Gambler("112", "Klint", "Eastwood",
LocalDate.of(1983, Month.FEBRUARY, 12));
        persons[13] = new Gambler("113", "Jane", "Fonda", LocalDate.of(1989,
Month.JANUARY, 13));

        Set<CardPlayerI> cardPlayers = new HashSet<>();

        for (int i = 0; i < persons.length - 1; i++) {
            for (int j = i + 1; j < persons.length; j++) {
                meet(persons[i], persons[j], cardPlayers);
            }
        }

        cardGame(cardPlayers);

        List<Entry<DicePlayerI, Integer>> sortedScores = new
LinkedList<>(diceScores.entrySet());
        Collections.sort(sortedScores, new Comparator<Entry<DicePlayerI,
Integer>>() {
            @Override
            public int compare(Entry<DicePlayerI, Integer> o1,
Entry<DicePlayerI, Integer> o2) {
                return o2.getValue().compareTo(o1.getValue());
            }
        }
        );
       for (Entry<DicePlayerI, Integer> entry:sortedScores) {
           System.out.println(entry.getKey()+"->"+entry.getValue());
       }
       System.out.println("");

        Map<CardPlayerI, ArrayList<Card>> opened = new LinkedHashMap<>();
        int maxAces = 0;
        for (CardPlayerI p : cardPlayers) {
            ArrayList<Card> cardsInHand = p.openCards();
            System.out.print(p + ":\n ");
            System.out.println(cardsInHand);
            opened.put(p, cardsInHand);
            int cAces = cntAces(cardsInHand);
            if (cAces > maxAces) {
                maxAces = cAces;
            }
        }

        ArrayList<CardPlayerI> winners = new ArrayList<>();
        for (CardPlayerI p : opened.keySet()) {
            int cAces = cntAces(opened.get(p));
            if (cAces == maxAces) {
```

```
                winners.add(p);
            }
        }
        System.out.println("----------------------------------------");
        System.out.println("winner in card Game: \n" + winners + " Aces: "
+ maxAces);
    }
}
```

Κώδικας 14.18 *Η κλάση BeckersMeeting*

Ας ξεκινήσουμε από την main της κλάσης. Καταρχάς, δημιουργούμε και φορτώνουμε στον πίνακα persons τα 14 πρόσωπα. Τα 8 είναι DicePlayer, τα 3 CardPlayer και τα υπόλοιπα 3 Gambler. Στη συνέχεια, ορίζουμε το cardPlayers τύπου HashSet. Το cardPlayers είναι ο κατάλογος στον οποίο εγγράφονται οι χαρτοπαίκτες σύμφωνα με την εκφώνηση. Η επιλογή του HashSet βασίστηκε στο γεγονός ότι κάθε χαρτοπαίκτης έχει νόημα να γραφτεί μόνο μία φορά. Αμέσως μετά υλοποιείται η απαίτηση να συναντηθούν τα πρόσωπα ανά δύο με τη βοήθεια της meet. Προσέξτε την εμφωλευμένη for μέσα στην οποία καλείται η meet. Οι συνδυασμοί ν αντικειμένων ανά 2 είναι ν×((ν-1))/2 , αυτό ακριβώς αποδίδει η for. Με άλλα λόγια, για κάθε ζεύγος προσώπων p1, p2 συναντάται το ζεύγος (p1, p2) μόνον εφόσον p1 διάφορο του p2, ενώ δεν επαναλαμβάνεται συνάντηση για το ζεύγος (p2, p1).

Αν μεταφερθούμε προσωρινά στη συνάρτηση meet, βλέπουμε πως καταρχάς ελέγχεται αν είναι και τα δύο πρόσωπα παίκτες ζαριού. Στην περίπτωση αυτή παίζουν μια παρτίδα ζάρια με την diceGame. Στη συνέχεια εφόσον είναι κάποιο από τα πρόσωπα χαρτοπαίκτης γίνεται η εγγραφή του στην ομάδα χαρτοπαικτών. Με το τέλος των συναντήσεων, διεξάγεται η παρτίδα χαρτιών με την cardGame.

Η diceGame ενημερώνει με το σκορ όλων των παιχνιδιών κάθε παίκτη την diceScores που είναι στατική μεταβλητή της κλάσης BeckersMeeting. Με την cardGame, ο κάθε χαρτοπαίκτης διατηρεί μια σειρά από κάρτες με βάση τις οποίες βγαίνουν οι νικητές.

Στη συνέχεια της main έχουμε να εμφανίσουμε τα αποτελέσματα. Θα πρέπει καταρχάς να ταξινομήσουμε τα αποτελέσματα των παιχνιδιών ζαριού ως προς φθίνουσα σειρά των σκορ. Εδώ μας βοηθάει η λίστα sortedScores. Κάθε στοιχείο αυτής της λίστας αναπαριστά ένα στοιχείο της diceScores. Στη συνέχεια ταξινομείται η λίστα με τη βοήθεια του Comparator που περνάμε ως παράμετρο στην Collections.sort. Προσέξτε πως ο Comparator δημιουργείται ως αντικείμενο ανώνυμης κλάσης. Επίσης, σημειώστε πως η compare του Comparator ταξινομεί κατά φθίνουσα σειρά. Έχοντας ταξινομημένα τα scores είναι εύκολο να τα εμφανίσουμε κατά φθίνουσα σειρά σύμφωνα με τις απαιτήσεις της εκφώνησης.

Τώρα θα πρέπει να ανοίξουν οι χαρτοπαίκτες τα χαρτιά που έχει ο κάθε ένας στο χέρι του, να γίνει καταμέτρηση και να ανακηρυχτεί ο νικητής ή οι νικητές. Οπότε κάθε ένας εγγεγραμμένος χαρτοπαίκτης ανοίγει τις κάρτες του που προσωρινά αποθηκεύονται στην τοπική μεταβλητή cardsInHand. Στη συνέχεια εμφανίζονται τα στοιχεία του παίκτη και τα χαρτιά του. Επιπλέον, τα χαρτιά κάθε παίκτη εισάγονται στην opened καθώς χρειάζονται για την τελική καταμέτρηση και ανάδειξη των νικητών. Παράλληλα με όλη αυτή τη διαδικασία υπολογίζεται ο μέγιστος αριθμός άσων που υπάρχουν στα χαρτιά κάθε χαρτοπαίκτη. Όλοι όσοι έχουν τον ίδιο αριθμό άσων θα πρέπει να ανακηρυχτούν νικητές.

Τέλος, εντοπίζονται οι νικητές της χαρτοπαιξίας και εμφανίζονται τα στοιχεία τους.

14.5.3 Πίνακας χημικών στοιχείων

Ο περιοδικός πίνακας των χημικών στοιχείων [3] περιλαμβάνει όλα τα γνωστά χημικά στοιχεία κατηγοριοποιημένα σε ομάδες, περιόδους και τομείς. Πιο συγκεκριμένα, υπάρχουν 18 ομάδες, 7 περίοδοι και 4 τομείς. Οι ομάδες αριθμούνται από 1 έως 18, οι περίοδοι από 1 έως 7, ενώ οι τομείς χαρακτηρίζονται με τους χαρακτήρες s, p, d, f. Κάθε χημικό στοιχείο ανήκει σε έναν τομέα, σε μία ομάδα και σε μία περίοδο. Επιπλέον, κάθε χημικό στοιχείο έχει ένα σύμβολο αναφοράς, π.χ. Η για το υδρογόνο, Na για το Νάτριο, καθώς και έναν ατομικό αριθμό.

Να αναπτυχθεί η κλάση ChemicalElements κάθε αντικείμενο της οποίας αναπαριστά ένα χημικό στοιχείο. Επιπλέον, να αναπτυχθεί η κλάση TableOfChemichalElements που αναπαριστά πίνακες χημικών στοιχείων και διαθέτει τις μεθόδους:

```
public boolean add(ChemicalElement cE)
```

Αν το στοιχείο cE, υπάρχει στον πίνακα επιστρέφει false. Αν δεν υπάρχει, το εισάγει και επιστρέφει true.

`public TableOfChemichalElements getGroup(int groupId)`
Επιστρέφει έναν πίνακα που περιλαμβάνει τα στοιχεία αυτού του πίνακα που ανήκουν στην ομάδα groupId.

`public TableOfChemichalElements getPeriod(int periodId)`
Επιστρέφει έναν πίνακα που περιλαμβάνει τα στοιχεία αυτού του πίνακα που ανήκουν στην περίοδο periodId.

`public TableOfChemichalElements getSection(char sectionId)`
Επιστρέφει έναν πίνακα που περιλαμβάνει τα στοιχεία αυτού του πίνακα που ανήκουν στον τομέα sectionId.

`public boolean isEmpty()`
Ελέγχει αν αυτός ο πίνακας είναι κενός.

`public Iterator groupIterator(int groupId)`
Επιστρέφει έναν Iterator στα στοιχεία της ομάδας με groupId. Ο Iterator υποστηρίζει εκτός από την hasNext και next και την remove. Η remove διαγράφει το τελευταίο στοιχείο που επέστρεψε η next. Δείτε τη διεπαφή της remove στην τεκμηρίωση του Iterator.
Προσθέστε main με βασικό τεστ κώδικα της κλάσης TableOfChemichalElements.

Λύση
Στον κώδικα 14.19, παρατίθεται η κλάση ChemicalElement.

```
public class ChemicalElement {

    private final String symbol;
    private final int atomicNumber;
    private final int group;
    private final int period;
    private final char section;

    public ChemicalElement(int group, int period, char sector, int
atomicNumber, String symbol) {
        this.group = group;
        this.period = period;
        this.section = sector;
        this.atomicNumber = atomicNumber;
        this.symbol = symbol;
    }

    public int getGroup() {
        return group;
    }

    public int getPeriod() {
        return period;
    }

    public char getSection() {
        return section;
    }

    public int getAtomicNumber() {
        return atomicNumber;
    }

    public String getSymbol() {
        return symbol;
    }
```

```
    @Override
    public int hashCode() {
        int hash = 7;
        hash = 89 * hash + this.symbol.hashCode();
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final ChemicalElement other = (ChemicalElement) obj;
        return this.symbol.equals(other.symbol);
    }

    @Override
    public String toString() {
        return symbol + ", AN=" + atomicNumber + ", group=" + group + ",
period=" + period + ", sector=" + section;
    }
}
```

Κώδικας 14.19 *Η κλάση ChemicalElement*

Η κλάση ChemicalElement είναι τυπική κλάση και περικλείει όλες τις αναγκαίες μεθόδους. Στον κώδικα 14.20 παρατίθεται η κλάση TableOfChemichalElements.

```
import java.util.ArrayList;
import java.util.Iterator;

public class TableOfChemichalElements {

    private final ArrayList<ChemicalElement> store = new ArrayList<>();

    public class GroupIterator implements Iterator {

        private final int groupId;
        private int current = 0;

        public GroupIterator(int groupId) {
            this.groupId = groupId;
        }

        @Override
        public boolean hasNext() {

            while (current < store.size()) {
                ChemicalElement cE = store.get(current);
                if (cE.getGroup() == groupId) {
                    return true;
                }
                current++;
            }
            return false;
```

```
        }

        @Override
        public Object next() {
            return store.get(current++);
        }

        @Override
        public void remove() {
            store.remove(--current);
        }
    }

    public Iterator groupIterator(int groupId) {
        return new GroupIterator(groupId);
    }

    public boolean add(ChemicalElement cE) {
        if (store.contains(cE)) {
            return false;
        }
        return store.add(cE);
    }

    public TableOfChemichalElements getGroup(int groupId) {
        TableOfChemichalElements rVal = new TableOfChemichalElements();
        for (ChemicalElement cE : store) {
            if (cE.getGroup() == groupId) {
                rVal.add(cE);
            }
        }
        return rVal;
    }

    public TableOfChemichalElements getPeriod(int periodId) {
        TableOfChemichalElements rVal = new TableOfChemichalElements();
        for (ChemicalElement cE : store) {
            if (cE.getPeriod() == periodId) {
                rVal.add(cE);
            }
        }
        return rVal;
    }

    public TableOfChemichalElements getSection(char sectionId) {
        TableOfChemichalElements rVal = new TableOfChemichalElements();
        for (ChemicalElement cE : store) {
            if (cE.getSection() == sectionId) {
                rVal.add(cE);
            }
        }
        return rVal;
    }

    public boolean isEmpty() {
        return store.isEmpty();
    }

    @Override
    public String toString() {
        StringBuilder t = new StringBuilder();
```

```
        for (ChemicalElement cE : store) {
            t.append(cE).append("\n");
        }
        return t.toString();
    }

    public static TableOfChemichalElements setUp() {
        ChemicalElement h = new ChemicalElement(1, 1, 's', 1, "H");
        ChemicalElement li = new ChemicalElement(1, 2, 's', 3, "Li");
        ChemicalElement na = new ChemicalElement(1, 3, 's', 11, "Na");
        ChemicalElement be = new ChemicalElement(2, 2, 's', 4, "Be");
        ChemicalElement mg = new ChemicalElement(2, 3, 's', 12, "Mg");
        ChemicalElement sc = new ChemicalElement(3, 4, 'd', 21, "Sc");
        ChemicalElement y = new ChemicalElement(3, 5, 'd', 39, "Y");
        ChemicalElement he = new ChemicalElement(18, 1, 'p', 2, "He");
        ChemicalElement la = new ChemicalElement(3, 6, 'f', 57, "La");
        TableOfChemichalElements gen = new TableOfChemichalElements();
        gen.add(h);
        gen.add(li);
        gen.add(na);
        gen.add(be);
        gen.add(mg);
        gen.add(sc);
        gen.add(y);
        gen.add(he);
        gen.add(la);
        return gen;
    }

    public static void main(String[] args) {

        TableOfChemichalElements gen = setUp();
        System.out.println(gen);
        TableOfChemichalElements group1 = gen.getGroup(1);
        System.out.println(group1);
        System.out.println(gen.getPeriod(1));
        System.out.println(gen.getSection('p'));

        System.out.println("----------------------------");
        Iterator it = gen.groupIterator(1);
        while (it.hasNext()) {
            ChemicalElement cE = (ChemicalElement) it.next();
            if (cE.getPeriod() == 2) {
                it.remove();
            }
            System.out.println(cE);
        }
        System.out.println("----------------------------");
        System.out.println(gen);
    }
}
```

Κώδικας 14.20 *Η κλάση TableOfChemichalElements*

Οι getGroup, getPeriod και getSection είναι προφανείς. Προκειμένου να υλοποιηθεί η getGroupIterator χρησιμοποιείται η εσωτερική κλάση GroupIterator η οποία υλοποιεί τη διεπαφή Iterator. Ο δημιουργός της κλάσης λαμβάνει ως παράμετρο το groupId της ομάδας του Iterator και ενημερώνει την αντίστοιχη μεταβλητή στιγμιότυπου. Επίσης, η ιδιωτική μεταβλητή current χρησιμεύει ως δείκτης στον πίνακα store της TableOfChemichalElements. Η hasNext αυξάνει την current έως ότου βρει στη θέση που υποδεικνύει η current στον πίνακα store στοιχείο που ανήκει στην ομάδα με groupId. Αν το βρει, επιστρέφει true διαφορετικά false. Η next επιστρέφει το στοιχείο στη θέση current του store και αυξάνει την current κατά 1. Η

remove διαγράφει το στοιχείο που επέστρεψε η τελευταία κλήση της next. Ωστόσο, η next αφού επιστρέψει το στοιχείο αυξάνει την current κατά 1. Επομένως, η remove για να διαγράψει το σωστό στοιχείο θα πρέπει να μειώσει την current κατά 1. Εξάλλου με τη διαγραφή ενός στοιχείου μειώνεται και το μέγεθος του store κατά 1.

Έχοντας την κλάση GroupIterator, η συνάρτηση getGroupIterator επιστρέφει απλώς ένα αντικείμενο GroupIterator. Η setUp είναι στατική συνάρτηση που απλώς φορτώνει μερικά στοιχεία σε κατάλληλο ArrayList και τα επιστρέφει. Επίσης, κατάλληλα έχει υπερφορτωθεί και η toString ώστε να τυπώνει τα στοιχεία του πίνακα ένα ανά γραμμή βασιζόμενη στην toString της ChemicalElement.

Στην main φορτώνουμε με τη βοήθεια της setUp τον πίνακα gen, στη συνέχεια ελέγχουμε τις getGroup, getPeriod και getSection. Τέλος, ελέγχεται ο Iterator.

14.5.4 Τρίλιζα

Να υλοποιηθεί το γνωστό παιχνίδι τρίλιζα κατά το οποίο ο χρήστης θα παίζει με τον υπολογιστή. Να δίνεται η δυνατότητα στον χρήστη να επιλέγει ως αντίπαλο έναν τεχνητό παίκτη που επιλέγει τις κινήσεις του με βάση κανόνες (Rule based) ή έναν τεχνητό παίκτη που βασίζεται στον αλγόριθμο MinMax [4]. Επιπλέον, να υποστηριχθεί η δυνατότητα, οι δύο τεχνητοί παίκτες να παίζουν μεταξύ τους.

Το ταμπλό του παιχνιδιού να εμφανίζεται όπως φαίνεται στην εικόνα 14.1 και να ανανεώνεται με κάθε κίνηση.

```
  0 1 2          0 1 2
--------       --------
0| | | |       0|X|O| |
--------       --------
1| | | |       1| | | |
--------       --------
2| | | |       2| | | |
--------       --------
```

Εικόνα 14.1 *Το ταμπλό της τρίλιζας στην αρχή του παιχνιδιού και μετά από 2 κινήσεις.*

Ο Rule based παίκτης να επιλέγει την κίνησή του με βάση τους ακόλουθους κανόνες:

1. Αν υπάρχει κίνηση με την οποία κάνει τρίλιζα την επιλέγει.
2. Αν ο αντίπαλός του έχει κίνηση με την οποία μπορεί να κάνει τρίλιζα στην επόμενη κίνηση, ο rule based επιλέγει αυτήν την κίνηση ώστε να μπλοκάρει τη νικηφόρα κίνηση του αντιπάλου.
3. Επιλέγει την πρώτη διαθέσιμη κίνηση.

Ο Minimax ως αλγόριθμος μπορεί να χρησιμοποιηθεί σε πολλά παιχνίδια για να εντοπίσει τη βέλτιστη κίνηση. Εκείνο που κάνει είναι ότι υπολογίζει το σύνολο των δυνατών παρτίδων και επιλέγει μονοπάτια που δεν οδηγούν σε ήττα. Ο τεχνητός παίκτης που βασίζεται στον Minimax είναι ένας τέλειος παίκτης και δεν χάνει ποτέ. Ωστόσο, αν αντιμετωπίσει έναν επίσης δυνατό παίκτη μπορεί να οδηγηθεί σε ισοπαλία. Ο αναγνώστης που ενδιαφέρεται για περισσότερες λεπτομέρειες μπορεί να παρακολουθήσει την εξαιρετική διάλεξη του Patrick Winston από το MIT [4].

Ο κώδικας να σχολιαστεί κατάλληλα ώστε να υποστηρίζει την παραγωγή τεκμηρίωσης java.

Λύση

Η λύση που παρατίθεται εδώ συντίθεται από επτά κλάσεις ως εξής: Η αφηρημένη κλάση Player και οι παράγωγές της σαφείς κλάσεις LPlayer μέσω της οποίας ο άνθρωπος παίζει τρίλιζα με τον υπολογιστή και οι τεχνητοί παίκτες RBPlayer (Rule based παίκτης), MinMaxPlayer (τεχνητός παίκτης που βασίζεται στον MiniMax αλγόριθμο), η βοηθητική κλάση Movement που αναπαριστά μία κίνηση στο ταμπλό, η κλάση Board που αναπαριστά το ταμπλό και η κλάση Game που οργανώνει τις υπόλοιπες και περιλαμβάνει την main της εφαρμογής.

Στη συνέχεια παρατίθεται ο κώδικας με κατάλληλα σχόλια από τα οποία αυτόματα μπορεί να εξαχθεί η τεκμηρίωση της εφαρμογής.

```
/**
 * Αναπαριστά το ταμπλό της Τρίλιζας
 *
 * @author Lefteris Moussiades
 */
public class Board {

    private final char[][] board;

    /**
     * Σημειώνει τις άδειες θέσεις στο ταμπλό
     */
    public static final char EMPTYCELL = ' ';

    /**
     * Ο Δημιουργός της κλάσης. Δημιουργεί ένα ταμπλό Τρίλιζας. Όλες οι
     * θέσεις αρχικοποιούνται στο EMPTYCELL
     */
    public Board() {
        board = new char[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                board[i][j] = EMPTYCELL;
            }
        }
    }

    /**
     * Δημιουργεί ένα αντίγραφο του ταμπλό
     *
     * @param source Το ταμπλό του οποίου θα δημιουργηθεί αντίγραφο
     */
    public Board(Board source) {
        board = new char[3][3];
        for (int i = 0; i < 3; i++) {
            System.arraycopy(source.board[i], 0, board[i], 0, 3);
        }
    }

    /**
     * Δημιουργεί αυτόματα μια κίνηση για τον παίκτη p. Καλείται μόνο εφόσον
     * υπάρχει μια μόνο θέση ελεύθερη στο ταμπλό και είναι σειρά του παίκτη
     * p.
     *
     */
    void autoMove(Player p) {
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == EMPTYCELL) {
                    board[i][j] = p.getId();
                }
            }
        }
    }

    /**
     * Εμφανίζει το ταμπλό
```

```
 */
void displayBoard() {
    System.out.println("  0 1 2");
    for (int i = 0; i < 3; i++) {
        System.out.println("--------");
        System.out.print(i);
        for (int j = 0; j < 3; j++) {
            System.out.print("|" + board[i][j]);
        }
        System.out.println("|");
    }
    System.out.println("--------");
}

/**
 * Θέτει την κίνηση mv του παίκτη p στο ταμπλό
 *
 */
public void setPlayerMove(Player p, Movement mv) {
    board[mv.getRow()][mv.getClmn()] = p.getId();
}

/**
 *
 * @return τον αριθμό των ελεύθερων θέσεων στο ταμπλό
 */
public int freePositions() {
    int cnt = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == EMPTYCELL) {
                cnt++;
            }
        }
    }
    return cnt;
}

private char lineWin(int ln) {
    if ((board[ln][0] != EMPTYCELL)
            && (board[ln][0] == board[ln][1])
            && (board[ln][0] == board[ln][2])) {
        return board[ln][0];
    }
    return EMPTYCELL;
}

private char columnWin(int clmn) {
    if ((board[0][clmn] != EMPTYCELL)
            && (board[0][clmn] == board[1][clmn])
            && (board[1][clmn] == board[2][clmn])) {
        return board[0][clmn];
    }
    return EMPTYCELL;
}

private char diagonalWin() {
    if ((board[0][0] != EMPTYCELL)
            && (board[0][0] == board[1][1])
            && (board[0][0] == board[2][2])) {
        return board[0][0];
```

```
        }
        if ((board[2][0] != EMPTYCELL)
                && (board[2][0] == board[1][1])
                && (board[2][0] == board[0][2])) {
            return board[2][0];
        }
        return EMPTYCELL;
    }

    /**
     *
     * @return την ταυτότητα του νικητή. Αν δεν υπάρχει νικητής επιστρέφει
     * EMPTYCELL;
     */
    public char win() {
        char rslt;
        for (int i = 0; i < 3; i++) {
            if ((rslt = lineWin(i)) != EMPTYCELL) {
                return rslt;
            }
            if ((rslt = columnWin(i)) != EMPTYCELL) {
                return rslt;
            }
        }
        return diagonalWin();
    }

    /**
     *
     * @return έναν πίνακα που περιέχει τις ελεύθερες θέσεις του ταμπλό
     */
    public Movement[] possibleMoves() {
        Movement[] rVal;
        int cnt = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == EMPTYCELL) {
                    cnt++;
                }
            }
        }
        if (cnt == 0) {
            return null;
        }
        rVal = new Movement[cnt];

        cnt = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == EMPTYCELL) {
                    rVal[cnt++] = new Movement(i, j);
                }
            }
        }
        return rVal;
    }
}
```

Κώδικας 14.21 *Η κλάση Board*

```java
/**
 * Αναπαριστά μία κίνηση στο ταμπλό της Τρίλιζας. Μία κίνηση αναπαρίσταται
 * από δύο ακέραιους, τη γραμμή και τη στήλη στο ταμπλό. Τα αντικείμενα της
 * κλάσης είναι αμετάβλητα.
 *
 * @author Lefteris Moussiades
 */
public class Movement {

    private final int r;
    private final int c;

    public Movement(int row, int clmn) {
        r = row;
        c = clmn;
    }

    public int getRow() {
        return r;
    }

    public int getClmn() {
        return c;
    }
}
```

Κώδικας 14.22 *Η κλάση Movement*

```java
/**
 * Αφηρημένη κλάση που αναπαριστά έναν παίκτη Τρίλιζας. Όλες οι σαφείς κλάσεις
 * που αναπαριστούν παίκτες Τρίλιζας πρέπει να είναι παράγωγες της Player.
 */
public abstract class Player {

    private final char id;
    private Player opponent;

    /**
     * Constructor
     *
     * @param id Η ταυτότητα του παίκτη, π.χ. x or o etc. Χρησιμοποιείται
     * επιπλέον κατά την επισήμανση των θέσεων του παίκτη στο ταμπλό
     *
     */
    public Player(char id) {
        this.id = id;
    }

    public char getId() {
        return id;
    }

    /**
     *
     * @param board
     * @return true Αν αυτός ο παίκτης έχει νικήσει
     */
    public boolean wins(Board board) {
        char wSign = board.win();
        return wSign == id;
```

```
    }

    /**
     * Ενημερώνει αυτόν τον παίκτη με τον αντίπαλό του. Ενημερώνει τον
     * αντίπαλο
     * με αυτόν τον παίκτη
     *
     * @param p the opponent Player
     */
    public void setOpponents(Player p) {
        opponent = p;
        p.opponent = this;
    }

    public Player getOpponent() {
        return opponent;
    }

    /**
     * Όλες οι σαφείς κλάσεις παικτών Τρίλιζας πρέπει να υλοποιούν αυτήν τη
     * μέθοδο
     *
     * @param par Λαμβάνει ως παράμετρο το ταμπλό της Τρίλιζας στην τρέχουσα
     * κατάσταση
     * @return Με βάση την τρέχουσα κατάσταση του ταμπλό, ο παίκτης
     * "αποφασίζει" την κίνησή του. Επιστρέφει αντικείμενο Movement
     */
    public abstract Movement move(Board par);

}
```

Κώδικας 14.23 *Η αφηρημένη κλάση Player*

```
/**
 * Represents ένα ζώντα παίκτη, δηλαδή έναν παίκτη που δίνει την κίνησή του από
 * το πληκτρολόγιο
 *
 */
public class LPlayer extends Player {

    public LPlayer(char id) {
        super(id);
    }

    /**
     * Λαμβάνει από το πληκτρολόγιο δύο ακέραιες τιμές
     *
     * @param board
     * @return Μια κίνηση (Movement)
     */
    @Override
    public Movement move(Board board) {
        Scanner cIn = new Scanner(System.in);
        int r, c;
        System.out.print("dwse grammi : ");
        r = cIn.nextInt();
        System.out.println();
        System.out.print("dwse stili : ");
        c = cIn.nextInt();
        return new Movement(r, c);
    }
```

```
}
```

Κώδικας 14.23 *Η κλάση LPlayer, παράγωγη της Player*

```
/**
 * It Represents a RuleBased Player
 *
 */
public class RBPlayer extends Player {

    public RBPlayer(char id) {
        super(id);
    }

    /**
     * Υλοποιεί την αφηρημένη μέθοδο της κλάσης Player σύμφωνα με τις
     * απαιτούμενες προδιαγραφές για τον Rule Based παίκτη
     *
     * @param board
     * @return Μία κίνηση (Movement)
     */
    @Override
    public Movement move(Board board) {
        Movement[] availM = board.possibleMoves();
        for (Movement mv : availM) {
            Board bd = new Board(board);
            bd.setPlayerMove(this, mv);
            if (getId() == bd.win()) {
                return mv;
            }
        }
        for (Movement mv : availM) {
            Board bd = new Board(board);
            bd.setPlayerMove(getOpponent(), mv);
            if (getOpponent().getId() == bd.win()) {
                return mv;
            }
        }
        return availM[0];
    }
}
```

Κώδικας 14.24 *Η κλάση RBPlayer, παράγωγη της Player*

```
/**
 * Κληρονομεί την αφηρημένη κλάση Player. Υλοποιεί τον αλγόριθμο minimax Ο
 * MiniMaxPlayer είναι τέλειος παίκτης. Δεν χάνει ποτέ.
 *
 * @author Lefteris
 */
public class MiniMaxPlayer extends Player {

    Movement currentMovement;

    public MiniMaxPlayer(char id) {
        super(id);
    }

    private int maxIdx(int[] tbl) {
        int rVal = 0, max = tbl[0];
        for (int i = 1; i < tbl.length; i++) {
```

```
            if (tbl[i] > max) {
                max = tbl[i];
                rVal = i;
            }
        }
        return rVal;
    }

    private int minIdx(int[] tbl) {
        int rVal = 0, min = tbl[0];
        for (int i = 1; i < tbl.length; i++) {
            if (tbl[i] < min) {
                min = tbl[i];
                rVal = i;
            }
        }
        return rVal;
    }

    private int minimax(Board board, Player p) {
        if (getOpponent().wins(board)) {
            return -1;
        }

        if (wins(board)) {
            return 1;
        }
        if (board.freePositions() == 0) {
            return 0;
        }
        Movement[] moves = board.possibleMoves();
        int[] scores = new int[moves.length];

        for (int i = 0; i < moves.length; i++) {
            Board newState = new Board(board);
            newState.setPlayerMove(p, moves[i]);
            scores[i] = minimax(newState, p.getOpponent());
        }

        if (this.equals(p)) {
            int maxScoreIdx = maxIdx(scores);
            currentMovement = moves[maxScoreIdx];
            return scores[maxScoreIdx];
        } else {
            int minScoreIdx = minIdx(scores);
            currentMovement = moves[minScoreIdx];
            return scores[minScoreIdx];
        }
    }

    /**
     *
     * @param board
     * @return Κίνηση με βάση τον minimax
     */
    @Override
    public Movement move(Board board) {
        minimax(board, this);
        return currentMovement;
    }
};
```

Κώδικας 14.25 *Η κλάση MiniMaxPlayer, παράγωγη της Player*

```
public class Game {

    Board board = new Board();
    Player p1;
    Player p2;

    /**
     * Δημιουργεί ένα παιχνίδι αποτελούμενο από ένα ταμπλό και δύο παίκτες
     * Ενημερώνει κάθε παίκτη με τον αντίπαλό του.
     *
     * @param p1 Player 1, αντίπαλος του Player 2
     * @param p2 Player 2, αντίπαλος του Player 1
     */
    Game(Player p1, Player p2) {
        this.p1 = p1;
        this.p2 = p2;
        p1.setOpponents(p2);
    }

    /**
     * Με βάση την board.win() ελέγχει αν υπάρχει νικητής
     *
     * @return Ο παίκτης που νίκησε ή null αν δεν υπάρχει νικητής
     * @see gr.ihu.cs.lmous.TicTacToe.Board#win()
     */
    Player winner() {
        char winnerSign = board.win();
        if (p1.getId() == winnerSign) {
            return p1;
        }
        if (p2.getId() == winnerSign) {
            return p2;
        }
        return null;
    }

    /**
     * Εκκινεί ένα παιχνίδι
     *
     * @param p Ο παίκτης που παίζει πρώτος
     * @return Τον νικητή ή null σε περίπτωση ισοπαλίας
     */
    Player play(Player p) {
        Player theWinner = null;
        Player current = p;

        do {
            Movement cell;
            if (board.freePositions() == 1) {
                board.autoMove(current);
            } else {
                cell = current.move(board);
                board.setPlayerMove(current, cell);
            }
            board.displayBoard();
            current = current.getOpponent();

        } while (board.freePositions() > 0
                && (theWinner = winner()) == null);
```

```
        return theWinner;
    }

    public static void main(String[] args)/*throws Exception*/ {
        Player p2 = new MiniMaxPlayer('X');
        Player p1 = new LPlayer('O');
        Player p3 = new RBPlayer('r');

        Game myGame = new Game(p2, p3);
        myGame.board.displayBoard();
        Player winner = myGame.play(p2);

        if (winner != null) {
            System.out.println("Winner is " + winner.getId());
        } else {
            System.out.println("withdraw ");
        }
    }
}
```

Κώδικας 14.26 *Η κλάση Game οργανώνει μια παρτίδα τρίλιζα*

14.6 Ασκήσεις προς Λύση

14.6.1 Πολλαπλές τράπουλες

Να υλοποιηθεί η κλάση AnyDeck, τα αντικείμενα της οποίας είναι τράπουλες που περιέχουν κάρτες στα όρια βαθμών (CardRank). Πιο συγκεκριμένα, ο δημιουργός της κλάσης έχει τη μορφή public AnyDeck(CardRank from, CardRank to) και δημιουργεί μια τράπουλα με όλες κάρτες από βαθμό from έως και βαθμό to. Να υλοποιηθεί η εφαρμογή της άσκησης 14.5.1 με τη βοήθεια δύο αντικειμένων της AnyDeck.

14.6.2 Μελέτη

Στην άσκηση 14.5.1, οι κλάσεις Deck52 και Deck32 έχουν πάρα πολλές ομοιότητες μεταξύ τους. Πράγματι, αν μελετήσετε τις δύο κλάσεις, θα διαπιστώσετε πως οι περισσότερες συναρτήσεις είναι σχεδόν ίδιες. Είναι εφικτό η μία κλάση να ορισθεί παράγωγη της άλλης ώστε να κληρονομήσει τις μεθόδους και να μην τις ξαναγράφουμε; Αν ναι, προχωρήστε σε αντίστοιχη τροποποίηση. Αν όχι, δώστε μια διαφορετική λύση που ενώ διατηρεί τις 2 κλάσεις, αποφεύγει την επανάληψη του κώδικα κατά το δυνατόν.

14.6.3 Περιοδικός Πίνακας

Υλοποιήστε την κλάση PeriodTable. Η κλάση αναπαριστά τον περιοδικό πίνακα των χημικών στοιχείων. Η κλάση για κάθε στοιχείο του περιοδικού πίνακα, περιέχει ένα αντικείμενο ChemicalElement που μπορούμε να προσπελάσουμε με τη συνάρτηση μέλος της κλάσης, ChemicalElement getElement(String symbol). Η κλάση δεν επιτρέπει εισαγωγές και διαγραφές στοιχείων, ενώ υλοποιεί Iterators, έναν για τις ομάδες, έναν για τις περιόδους και έναν για τους τομείς.

Βιβλιογραφία

[1] “Interfaces (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance).” https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html (accessed Oct. 27, 2021).

[2] “Multiple Inheritance of State, Implementation, and Type (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance).” https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html (accessed Oct. 27, 2021).

[3] “Περιοδικός πίνακας των χημικών στοιχείων,” Βικιπαίδεια. Oct. 23, 2021. Accessed: Nov. 02, 2021. [Online]. Available: https://el.wikipedia.org/w/index.php?title Περιοδικός_πίνακας_των_χημικών_στοιχείων&oldid=9105038

[4] “Minimax,” Wikipedia. Oct. 05, 2021. Accessed: Nov. 01, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1048398851

[5] MIT OpenCourseWare, 6. Search: Games, Minimax, and Alpha-Beta, (Jan. 10, 2014). Accessed: Nov. 01, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=STjW3eH0Cik

Κεφάλαιο 15

Σύνοψη
Στην ενότητα αυτή παρουσιάζεται η βασική διαχείριση των αρχείων. Πιο συγκεκριμένα, παρουσιάζεται η ανάγνωση δεδομένων με την Scanner, η ανάγνωση και εγγραφή σε δομημένα αρχεία, η ανάγνωση και εγγραφή σε αρχεία ψηφιολέξεων, η διαχείριση των αρχείων τυχαίας προσπέλασης, η σειριοποίηση και διάφορες χρήσιμες προκαθορισμένες λειτουργίες που σχετίζονται με αρχεία.

Προαπαιτούμενη γνώση
Η ενσωμάτωση, η κληρονομικότητα και ο πολυμορφισμός στο αντικειμενοστρεφές μοντέλο με Java, οι αφηρημένες κλάσεις καθώς και οι βασικές προκαθορισμένες κλάσεις, δηλαδή οι String, StringBuilder, ArrayList, LinkedList, Stack, PriorityQueue, HashSet, HashMap, HashTree, LocalDate, Localtime, Period και Duration. Οι διεπαφές στην Java.

Λέξεις κλειδιά
Είσοδος (input), έξοδος (output), τυχαία προσπέλαση (random access), σειριοποίηση (serialization)

15 Διαχείριση Αρχείων

Η διαχείριση των αρχείων αποτελεί σημαντική δυνατότητα κάθε προγραμματιστικού περιβάλλοντος. Η Java παρέχει δύο πακέτα για τη διαχείριση των αρχείων. Το πακέτο java.io και το java.nio. Σε γενικές γραμμές, η λειτουργικότητα που προσφέρουν είναι περίπου η ίδια. Το πακετο java.io παρέχει τον καθιερωμένο τρόπο ανάγνωσης και εγγραφής σε αρχεία. Το πακέτο java.nio προστέθηκε στην 4η έκδοση της Java ως εναλλακτική λύση με στόχο να προσφέρει ταχύτερες λειτουργίες ανάγνωσης και εγγραφής. Η κύρια διαφορά μεταξύ των δύο πακέτων είναι πως το μεν java.io παρέχει συγχρονισμένες λειτουργίες (synchronous IO ή blocking IO) ανάγνωσης και εγγραφής, το δε java.nio παρέχει ασύγχρονες λειτουργίες (asynchronous IO ή non-blocking IO). Αυτό σημαίνει πως όταν ένα πρόγραμμα επιχειρήσει να διαβάσει ή να γράψει δεδομένα σε αρχείο, χρησιμοποιώντας το java.io, τότε η εκτέλεσή του σταματά έως ότου ολοκληρωθεί η λειτουργία ανάγνωσης ή εγγραφής. Όταν όμως χρησιμοποιείται το java.nio, τότε η εκτέλεση του προγράμματος συνεχίζεται παράλληλα με τη λειτουργία ανάγνωσης ή εγγραφής.

Στην ενότητα αυτή παρουσιάζονται βασικές κλάσεις του πακέτου java.io. Το πακέτο περιέχει σημαντικό αριθμό κλάσεων και διεπαφών. Εδώ συζητάμε ένα μέρος. Ωστόσο, οι κλάσεις και οι τεχνικές που παρουσιάζονται σε αυτήν την ενότητα μπορούν να αξιοποιηθούν ώστε να καλύψουν τις περισσότερες ανάγκες που προκύπτουν κατά την είσοδο και έξοδο των δεδομένων στις εφαρμογές.

Το πακέτο java.io βασίζεται στην έννοια των ροών εισόδου-εξόδου (Input/Output streams). Μια ροή είναι μια γραμμή επικοινωνίας μεταξύ του προγράμματος και μιας συσκευής εισόδου-εξόδου. Μέσα από τη ροή μεταφέρονται δεδομένα μεταξύ συσκευής και προγράμματος. Το πακέτο java.io υποστηρίζει τρεις βασικούς τύπους ροών:

- Ροές ψηφιολέξης (Byte Streams): Ροές μέσα από τις οποίες τα δεδομένα μεταφέρονται ψηφιολέξη προς ψηφιολέξη. Στο πακέτο java.io, όλες οι κλάσεις που υποστηρίζουν ροές ψηφιολέξεων είναι παράγωγες των αφηρημένων κλάσεων InputStream ή OutputStream.
- Ροές χαρακτήρων (Character Streams): Ροές μέσα από τις οποίες τα δεδομένα μεταφέρονται ως σειρά χαρακτήρων. Στο πακέτο java.io, όλες οι κλάσεις που υποστηρίζουν ροές χαρακτήρων είναι παράγωγες των αφηρημένων κλάσεων Reader ή Writer. Οι ροές χαρακτήρων χρησιμοποιούν τις ροές ψηφιολέξεων για να πραγματοποιήσουν είσοδο ή έξοδο αλλά υποστηρίζουν επιπλέον αυτόματη κωδικοποίηση χαρακτήρων (character encoding).
- Ροές προσωρινής αποθήκευσης (Buffered streams): Οι ροές προσωρινής αποθήκευσης αξιοποιούνται ως περιβλήματα (wrappers) για τις ροές ψηφιολέξεων και χαρακτήρων. Οι ροές ψηφιολέξεων και οι ροές χαρακτήρων διαβάζουν ή γράφουν μία ψηφιολέξη ή έναν χαρακτήρα κάθε φορά. Επομένως, οι λειτουργίες ανάγνωσης και εγγραφής είναι σημαντικά χρονοβόρες. Αντίθετα, αν ζητήσουμε από μία ροή προσωρινής αποθήκευσης να μας φέρει έναν χαρακτήρα από συσκευή εισόδου, η ροή θα διαβάσει έναν σημαντικό αριθμό χαρακτήρων, θα τους αποθηκεύσει

στη μνήμη και θα μας επιστρέψει τον χαρακτήρα που ζητήσαμε. Έτσι, στην επόμενη ανάγνωση δεν είναι απαραίτητη η πρόσβαση στη συσκευή εισόδου καθώς αρκετοί από τους επόμενους χαρακτήρες βρίσκονται ήδη στη μνήμη. Συμπερασματικά, οι ροές προσωρινής αποθήκευσης επιταχύνουν σημαντικά τις λειτουργίες ανάγνωσης και εγγραφής.

Πέραν των ανωτέρω, στην 5η έκδοση της Java προστέθηκε η κλάση Scanner που ανήκει στο πακέτο java.util την οποία έχουμε ήδη χρησιμοποιήσει για είσοδο από το πληκτρολόγιο και η οποία μπορεί να διασυνδεθεί και με άλλες συσκευές εισόδου. Γενικά, η κλάση χρησιμοποιείται εκτεταμένα και για αυτό η παρουσίαση ξεκινά με την Scanner.

15.1. Είσοδος με την Scanner

Με την Scanner έχουμε τη δυνατότητα να διαβάζουμε δεδομένα που αναπαριστούν θεμελιώδεις τύπους ή συμβολοσειρές [1]. Ένα αντικείμενο Scanner διασπά την είσοδό του σε επιμέρους τμήματα σύμφωνα με κάποιο διαχωριστικό (delimiter) που εξ ορισμού είναι ένας λευκός χαρακτήρας. Η ανάγνωση των δεδομένων γίνεται με τις ποικίλες μεθόδους hasNext και next που διαθέτει η Scanner, π.χ. hasNextDouble, nextDouble, hasNextInt, nextInt.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Locale;
import java.util.Scanner;

public class ScannerDemo {

    private static final String DOUBLEDATA =
"data"+File.separator+"doubledata.dat";

    public static void createDoubleData() throws IOException {
        PrintWriter out = new PrintWriter(DOUBLEDATA);
        for (double d = 0.5; d < 10; d += 0.4) {
            if (d >= 5 && d < 5.5) {
                out.println();
            }
            out.printf(Locale.US, "%5.1f", d);
        }
        out.close();
    }

    public static void readDoubles() throws FileNotFoundException {
        File file = new File(DOUBLEDATA);
        Scanner scanner = new Scanner(file);
        scanner.useLocale(Locale.US);
        while (scanner.hasNextDouble()) {
            System.out.println(scanner.nextDouble());
        }
        scanner.close();
    }

    public static void readLines() throws FileNotFoundException {
        File file = new File(DOUBLEDATA);
        Scanner scanner = new Scanner(file);
        scanner.useLocale(Locale.US);
        while (scanner.hasNextLine()) {
            System.out.println(scanner.nextLine());
        }
        scanner.close();
```

```
        }

        public static void main(String[] args) throws IOException {
            createDoubleData();
            readDoubles();
            readLines();
        }
    }
```

Κώδικας 15.1 *Επίδειξη ανάγνωσης με την Scanner.*

Σε αυτό το παράδειγμα, η ανάγνωση των δεδομένων γίνεται από ένα αρχείο με όνομα doubledata.dat που βρίσκεται στον φάκελο data. Ο δε φάκελος data είναι τοποθετημένος στον φάκελο εργασίας, δηλαδή στον γονικό φάκελο του project στον οποίο περιλαμβάνεται η κλάση ScannerDemo. Η σχετική αυτή διαδρομή αποθηκεύεται στη σταθερά DOUBLEDATA. Η σταθερά αυτή θα μπορούσε να οριστεί ως "data"+"\\"+"doubledata.dat", καθώς το διαχωριστικό στα ονόματα των διαδρομών αρχείων είναι ο χαρακτήρας "\". Αν επομένως διαμορφώσετε την DOUBLEDATA κατ' αυτόν τον τρόπο, η εφαρμογή σας θα τρέχει σωστά. Ωστόσο, η κωδικοποίηση αυτή δεν είναι ικανοποιητική για την Java που είναι διαπλατφορμική γλώσσα. Προτιμήστε τη διαμόρφωση που αξιοποιεί την File.separator και αφήστε την Java να επιλέξει το κατάλληλο διαχωριστικό ανάλογα με το λειτουργικό που θα τρέξει η εφαρμογή σας.
Το αρχείο doubledata.dat περιέχει τα ακόλουθα δεδομένα.

```
0.5  0.9  1.3  1.7  2.1  2.5  2.9  3.3  3.7  4.1  4.5  4.9
5.3  5.7  6.1  6.5  6.9  7.3  7.7  8.1  8.5  8.9  9.3  9.7
```

Πρόκειται για μια σειρά από πραγματικούς αριθμούς. Μπορείτε να δημιουργήσετε το αρχείο με κλήση της createDoubleData του κώδικα 15.1 ή χρησιμοποιώντας έναν συντάκτη κειμένου (text editor).

Η readDoubles διαβάζει τα δεδομένα του αρχείου αριθμό προς αριθμό. Μέσα στη συνάρτηση καταρχάς ορίζουμε ένα αντικείμενο της κλάσης File. Η κλάση File διευκολύνει στη διαχείριση των αρχείων. Μπορείτε να δείτε περισσότερα για την κλάση στην ενότητα 15.6. Στη συνέχεια, δημιουργούμε ένα αντικείμενο Scanner που συνδέεται με το συγκεκριμένο αρχείο. Η scanner.useLocale(Locale.US) έχει ως σκοπό να διασφαλίσει πως οι πραγματικοί αριθμοί θα μεταφραστούν από την Scanner θεωρώντας πως το διαχωριστικό των δεκαδικών είναι η τελεία και των χιλιάδων το κόμμα. Βέβαια, στο παράδειγμά μας δεν βλέπουμε το διαχωριστικό των χιλιάδων, καθώς τα δεδομένα μας είναι αριθμοί μικρότεροι του 10. Στη συνέχεια, η scanner ενόσω διαπιστώνει ύπαρξη πραγματικού αριθμού στη ροή εισόδου με την hasNextDouble, τον διαβάζει με την nextDouble. Κάθε είσοδος διαχωρίζεται από την επόμενη από την ύπαρξη ενός λευκού χαρακτήρα που στο παράδειγμά μας συνήθως είναι ένα διάστημα με εξαίρεση τον λευκό χαρακτήρα μεταξύ 4.9 και 5.3 που είναι ο χαρακτήρας newline.

Η readLines με παρόμοια τεχνική, διαβάζει από τη ροή εισόδου, μία γραμμή κάθε φορά, δηλαδή διαβάζει τα δεδομένα έως ότου συναντήσει τον χαρακτήρα newline. Κάθε γραμμή την επιστρέφει ως συμβολοσειρά. Οι ενδιάμεσοι λευκοί χαρακτήρες συμπεριλαμβάνονται στην επιστρεφόμενη συμβολοσειρά, όχι όμως και ο χαρακτήρας newline.

Προσέξτε πως οι συναρτήσεις του κώδικα 15.1 έχουν στη διεπαφή τους τη λέξη-κλειδί throws ακολουθούμενη από τύπους εξαιρέσεων. Αυτό οφείλεται σε πιθανά λάθη που μπορεί να προκύψουν κατά τη διαχείριση αρχείων. Γενικά σε αυτήν την ενότητα, τα πιθανά λάθη αντιμετωπίζονται με αυτόν τον τρόπο. Στην επόμενη ενότητα (κεφάλαιο 16) θα μιλήσουμε αναλυτικά για τις τεχνικές διαχείρισης των λαθών και θα δούμε περισσότερο σύνθετους και πιο αποτελεσματικούς τρόπους διαχείρισής τους.

Τέλος, προσέξτε πως κάθε συνάρτηση που αρχικοποιεί μία ροή την κλείνει κιόλας με την κλήση της close. Η ενέργεια αυτή είναι υποχρεωτική ώστε να διασφαλίζεται η ορθότητα των δεδομένων. Εξαίρεση αποτελεί η περίπτωση κατά την οποία η Scanner συνδέεται με την System.in όπως επισημάνθηκε στην ενότητα 4.3.

15.2 Δομημένα αρχεία δεδομένων

Πολύ συχνά, οι εφαρμογές μας διαχειρίζονται αρχεία στα οποία τα δεδομένα έχουν συγκεκριμένη δομή. Ας υποθέσουμε πως έχουμε μια σειρά από μαθητές των οποίων τα δεδομένα θέλουμε να αποθηκεύουμε σε ένα

αρχείο ή να τα διαβάζουμε από ένα αρχείο. Καταρχάς, θα πρέπει κάπως να αναπαριστούμε τα δεδομένα αυτά στη μνήμη του προγράμματός μας. Ο τυπικός τρόπος είναι ο σχεδιασμός κατάλληλης κλάσης. Έστω λοιπόν η κλάση Student, παράγωγη της Person.

```
import java.time.LocalDate;
import java.time.Month;
import java.util.Locale;

public class Student extends Person {

    private final double[] scores = new double[Subject.values().length];
    public final static int RECSIZE = 69;

    public Student(String id, String fName, String sName, LocalDate
birthday) {
        super(id, fName, sName, birthday);
    }

    public void setScore(Subject subject, double score) {
        scores[subject.ordinal()] = score;
    }

    public void setScores(double[] scores) {
        System.arraycopy(scores, 0, this.scores, 0, scores.length <
this.scores.length ? scores.length : this.scores.length);
    }

    public double getScore(Subject subject) {
        return scores[subject.ordinal()];
    }

    @Override
    public String toString() {
        return getId() + " " + getfName() + " " + getsName()
                + String.format(Locale.US, "%5.1f", getScore(Subject.MATH))
                + String.format(Locale.US, "%5.1f",
getScore(Subject.HISTORY))
                + String.format(Locale.US, "%5.1f",
getScore(Subject.LANGUAGE))
                + String.format(Locale.US, "%5.1f",
getScore(Subject.CHEMISTRY))
                + String.format(Locale.US, "%5.1f",
getScore(Subject.PHYSICS))
                + String.format(Locale.US, "%5.1f",
getScore(Subject.BIOLOGY))
                + " " + getBirthday();
    }

    public String toFixedWidth() {
        return String.format("%4s", getId())
                + String.format("%15s", getfName())
                + String.format("%15s", getsName())
                + String.format(Locale.US, "%4.1f", getScore(Subject.MATH))
                + String.format(Locale.US, "%4.1f",
getScore(Subject.HISTORY))
                + String.format(Locale.US, "%4.1f",
getScore(Subject.LANGUAGE))
                + String.format(Locale.US, "%4.1f",
getScore(Subject.CHEMISTRY))
                + String.format(Locale.US, "%4.1f",
getScore(Subject.PHYSICS))
```

```
                + String.format(Locale.US, "%4.1f",
    getScore(Subject.BIOLOGY))
                + getBirthday();
        }
    }
```

Κώδικας 15.2 *Η κλάση Student.*

Κάθε αντικείμενο Student εκτός από τα χαρακτηριστικά που κληρονομεί από την Person έχει και έναν πίνακα scores στον οποίο αποθηκεύονται οι βαθμοί του. Οι βαθμοί ελέγχονται με τη βοήθεια του απαριθμήσιμου τύπου Subject.

```
public enum Subject {
     MATH, HISTORY, LANGUAGE, CHEMISTRY, PHYSICS, BIOLOGY;
}
```

Ο δημιουργός της κλάσης καλεί απλώς τον δημιουργό της γονικής, ενώ οι μέθοδοι setScores, setScore και getScore χρησιμεύουν ως αναγνώστες και ρυθμιστές για τους βαθμούς του μαθητή.

Αν υποθέσουμε τώρα πως θέλουμε να γράψουμε ή να διαβάσουμε τα δεδομένα μιας ομάδας μαθητών, έχουμε δύο ειδών τεχνικές στη διάθεσή μας.

1. Τα δεδομένα ενός μαθητή διαχωρίζονται μεταξύ τους με κατάλληλο χαρακτήρα. Στην περίπτωσή μας, μπορούμε να χρησιμοποιήσουμε ένα διάστημα. Ωστόσο, αν η κλάση Student διέθετε μια μεταβλητή, π.χ. name στην οποία εκχωρούνταν τιμές που περιλαμβάνουν διάστημα, π.χ. “John Black”, τότε είναι προφανές πως θα έπρεπε να χρησιμοποιήσουμε άλλο χαρακτήρα διαχωρισμού. Τα αρχεία που διαμορφώνονται σύμφωνα με αυτήν την τεχνική ονομάζονται οριοθετημένα αρχεία (delimited files).
2. Κάθε πεδίο από τα δεδομένα ενός μαθητή εγγράφεται σε προκαθορισμένο πλάτος. Για παράδειγμα, κάθε βαθμός μπορεί να εγγραφεί σε πλάτος 4 χαρακτήρων καθώς οι βαθμοί μας παρακολουθούνται με ακρίβεια 1 δεκαδικού και δεν υπερβαίνουν τις 2 ακέραιες θέσεις. Μία θέση επιπλέον χρειάζεται για το διαχωριστικό των δεκαδικών, π.χ. 19.5. Το όνομα μπορεί να εγγραφεί σε πλάτος 15 χαρακτήρων εφόσον εκτιμούμε ότι στα δεδομένα μας δεν υπάρχει όνομα που να υπερβαίνει τους 15 χαρακτήρες. Η τεχνική αυτή ονομάζεται οριοθέτηση σταθερού πλάτους (fixed width).

Η συνάρτηση toString της κλάσης Student επιστρέφει μια String αναπαράσταση του αντικειμένου Student όπου κάθε επιμέρους στοιχείο διαχωρίζεται από το προηγούμενο με ένα διάστημα. Επομένως, ένα αντικείμενο Student μπορεί να γραφεί σε ροή εξόδου σύμφωνα με την πρώτη τεχνική οριοθέτησης χωρίς καμιά περαιτέρω επεξεργασία.

Η συνάρτηση toFixedWidth επιστρέφει μια String αναπαράσταση του Student με βάση την οριοθέτηση σταθερού πλάτους.

Ο προσδιοριστής διαμόρφωσης που λαμβάνει η String.format υπακούει στους ίδιους κανόνες με αυτόν που παρουσιάστηκε στην ενότητα 4.2.1 για την printf.

Στη συνέχεια, παρουσιάζεται πρώτα ένα παράδειγμα που γράφει και διαβάζει στοιχεία μαθητών σε οριοθετημένο αρχείο και ένα δεύτερο παράδειγμα που γράφει και διαβάζει σε αρχεία σταθερού πλάτους.

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;
import java.util.Scanner;

public class StudentsDelimited {
```

```
    static void createStudents(String fName) throws IOException {
        File file = new File(fName);
        BufferedWriter out = new BufferedWriter(new FileWriter(file));
        Student jim = new Student("113", "Jim", "Blue", LocalDate.of(2005,
Month.APRIL, 1));
        jim.setScores(new double[]{10, 11, 12, 13, 14, 15});
        Student jack = new Student("212", "Jack", "Green", LocalDate.of(2005,
Month.MAY, 1));
        jack.setScores(new double[]{11, 12, 13, 14, 15, 16});
        Student mary = new Student("111", "Mary", "Red", LocalDate.of(2005,
Month.JUNE, 1));
        mary.setScores(new double[]{12, 13, 14, 15, 16, 17});
        Student kely = new Student("201", "Kely", "Green", LocalDate.of(2005,
Month.JULY, 1));
        kely.setScores(new double[]{13, 14, 15, 16, 17, 18});
        out.write(jim + "\n");
        out.write(jack + "\n");
        out.write(mary + "\n");
        out.write(kely + "\n");

        out.close();
    }

    static ArrayList<Student> readStudents(String fName) throws
FileNotFoundException  {
        ArrayList<Student> rVal = new ArrayList<>();
        File input = new File(fName);
        Scanner scanner = new Scanner(input);
        while (scanner.hasNextLine()) {
            String st = scanner.nextLine();
            String[] stFields = st.split(" ");
            Student student = new Student(stFields[0], stFields[1],
stFields[2], LocalDate.parse(stFields[9]));
            student.setScore(Subject.MATH, Double.parseDouble(stFields[3]));
            student.setScore(Subject.HISTORY,
Double.parseDouble(stFields[4]));
            student.setScore(Subject.LANGUAGE,
Double.parseDouble(stFields[5]));
            student.setScore(Subject.CHEMISTRY,
Double.parseDouble(stFields[6]));
            student.setScore(Subject.PHYSICS,
Double.parseDouble(stFields[7]));
            student.setScore(Subject.BIOLOGY,
Double.parseDouble(stFields[8]));
            rVal.add(student);
        }
        scanner.close();
        return rVal;
    }

    public static void main(String[] args) throws IOException {
        String fName = "data" + File.separator + "studentsd.dat";
        createStudents(fName);
        ArrayList<Student> students = readStudents(fName);
        System.out.println(students);
    }
}
```

Κώδικας 15.3 *Ανάγνωση και εγγραφή σε οριοθετημένο αρχείο*

Στον κώδικα 15.3 η συνάρτηση createStudents δημιουργεί έναν αριθμό από αντικείμενα Students και τα αποθηκεύει σε αρχείο. Το αρχείο το διαχειριζόμαστε μέσω ενός FileWriter που είναι κλάση κατάλληλη για να γράφουμε δεδομένα με τη μορφή χαρακτήρων. Επιπλέον χρησιμοποιείται ως περίβλημα ένα αντικείμενο BufferedWriter για επιτάχυνση της διαδικασίας εγγραφής. Κάθε μαθητής εισάγεται στο αρχείο ακολουθούμενος από τον χαρακτήρα newline. Μετά την εκτέλεση της createStudents, το αρχείο 'studentsd.dat' διαμορφώνεται ως ακολούθως:

```
113 Jim Blue 10.0 11.0 12.0 13.0 14.0 15.0 2005-04-01
212 Jack Green 11.0 12.0 13.0 14.0 15.0 16.0 2005-05-01
111 Mary Red 12.0 13.0 14.0 15.0 16.0 17.0 2005-06-01
201 Kely Green 13.0 14.0 15.0 16.0 17.0 18.0 2005-07-01
```

Η readStudents επιστρέφει ένα ArrayList από αντικείμενα Student που φορτώνει από το 'studentsd.dat'. Για να επιτύχει τη φόρτωση χρησιμοποιεί ένα αντικείμενο Scanner με το οποίο διαβάζει τις γραμμές του αρχείου μια προς μια. Η γραμμή εύκολα αποσυντίθεται στα επιμέρους στοιχεία της με την split. Στη συνέχεια, γνωρίζοντας τη γραμμογράφηση του αρχείου, μετατρέπει τα επιμέρους στοιχεία της γραμμής στους κατάλληλους τύπους και δημιουργεί το αντικείμενο Student που προσθέτει στο ArrayList που επιστρέφει. Τέλος, κλείνει τον Scanner. Στην main περιλαμβάνεται βασικός κώδικας ελέγχου.

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;
import java.util.Scanner;

public class StudentsToStdWidth {

    static void createStudents(String fName) throws IOException {
        File file = new File(fName);
        BufferedWriter out = new BufferedWriter(new FileWriter(file));
        Student jim = new Student("113", "Jim", "Blue", LocalDate.of(2005,
Month.MARCH, 1));
        jim.setScores(new double[]{10, 11, 12, 13, 14, 15});
        Student jack = new Student("212", "Jack", "Green",
LocalDate.of(2005, Month.MARCH, 2));
        jack.setScores(new double[]{11, 12, 13, 14, 15, 16});
        Student mary = new Student("911", "Mary", "Red", LocalDate.of(2005,
Month.MARCH, 3));
        mary.setScores(new double[]{12, 13, 14, 15, 16, 17});
        Student kely = new Student("201", "Kely", "Green",
LocalDate.of(2005, Month.MARCH, 4));
        kely.setScores(new double[]{13, 14, 15, 16, 17, 18});
        out.write(jim.toFixedWidth() + "\n");
        out.write(jack.toFixedWidth() + "\n");
        out.write(mary.toFixedWidth() + "\n");
        out.write(kely.toFixedWidth() + "\n");
        out.close();
    }

    public static Student toStudent(String st) {
        Student student = new Student(st.substring(0, 4).trim(),
                st.substring(4, 19).trim(),
                st.substring(19, 34).trim(),
                LocalDate.parse(st.substring(58))
```

```
            );
            student.setScore(Subject.MATH, Double.parseDouble(st.substring(34,
38).trim()));
            student.setScore(Subject.HISTORY,
Double.parseDouble(st.substring(38, 42).trim()));
            student.setScore(Subject.LANGUAGE,
Double.parseDouble(st.substring(42, 46).trim()));
            student.setScore(Subject.CHEMISTRY,
Double.parseDouble(st.substring(46, 50).trim()));
            student.setScore(Subject.PHYSICS,
Double.parseDouble(st.substring(50, 54).trim()));
            student.setScore(Subject.BIOLOGY,
Double.parseDouble(st.substring(54, 58).trim()));
            return student;
        }

        static ArrayList<Student> readStudents(String fName) throws
FileNotFoundException {
            ArrayList<Student> rVal = new ArrayList<>();
            File input = new File(fName);
            Scanner scanner = new Scanner(input);
            while (scanner.hasNextLine()) {
                String st = scanner.nextLine();
                Student student = toStudent(st);
                rVal.add(student);
            }
            scanner.close();
            return rVal;
        }

        public static void main(String[] args) throws IOException {
            createStudents("data" + File.separator + "students.dat");
            ArrayList<Student> students = readStudents("data" + File.separator
+ "students.dat");
            System.out.println(students);
        }
}
```

Κώδικας 15.4 *Ανάγνωση και εγγραφή σε αρχείο σταθερού πλάτους*

Στον κώδικα 15.4 επιδεικνύεται η ανάγνωση και εγγραφή σε αρχείο σταθερού πλάτους. Μετά από εκτίμηση του μέγιστου πιθανού πλάτους των δεδομένων ενός Student σχεδιάστηκε η ακόλουθη γραμμογράφηση του αρχείου. Το id τοποθετείται σε πλάτος 4 χαρακτήρων, το όνομα σε πλάτος 15, το επώνυμο σε πλάτος 15, οι βαθμοί σε πλάτος 4 με ακρίβεια 1 δεκαδικού. Τέλος, η ημερομηνία γέννησης γράφεται και διαβάζεται με την εξ ορισμού διαμόρφωσή της που καταλαμβάνει πλάτος 10 (yyyy-mm-dd). Η γραμμογράφηση αυτή επιστρέφεται από τη συνάρτηση toStandardWidth της κλάσης Student. Επομένως, η createStudents, το μόνο που κάνει είναι να δημιουργεί μια σειρά από αντικείμενα Student και να ενημερώνει το αρχείο students.dat. Η εγγραφή στο αρχείο γίνεται μέσω ενός FileWriter που είναι η κατάλληλη κλάση για εγγραφή χαρακτήρων σε αρχεία. Για επιτάχυνση της διαδικασίας εγγραφής χρησιμοποιείται ένα BufferedWriter ως περίβλημα του αντικειμένου FileWriter.
Μετά την εκτέλεση της createStudent, το αρχείο students.dat έχει ως εξής:

```
113           Jim            Blue10.011.012.013.014.015.02005-03-01
212          Jack           Green11.012.013.014.015.016.02005-03-02
911          Mary             Red12.013.014.015.016.017.02005-03-03
201          Kely           Green13.014.015.016.017.018.02005-03-04
```

Στη συνέχεια, η readStudents διαβάζει μία προς μία τις γραμμές του αρχείου και τις μετατρέπει σε αντικείμενα Student με τη βοήθεια της toStudent. Είναι προφανές πως η ανάγνωση τόσο οριοθετημένων

αρχείων όσο και αρχείων σταθερού πλάτους προϋποθέτει τη γνώση της δομής τους. Στην main παρατίθεται βασικός κώδικας ελέγχου.

15.3 Ανάγνωση και εγγραφή σε αρχεία ψηφιολέξεων

Είδαμε πως μπορούμε να γράφουμε και να διαβάζουμε χαρακτήρες από ένα αρχείο. Μερικές φορές όμως τα αρχεία που θέλουμε να επεξεργαστούμε δεν είναι αρχεία κειμένου ώστε να περιέχουν χαρακτήρες. Για παράδειγμα, έστω ότι θέλουμε να αντιγράψουμε ένα αρχείο εικόνας. Σε αυτήν την περίπτωση θα πρέπει να διασυνδεθούμε μέσω ροής ψηφιολέξεων.

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class BinaryData {

    private static final String INPUT = "data" + File.separator +
"sunset.jpg";
    private static final String OUTPUT = "data" + File.separator +
"sunsetcopy.jpg";
    private static final String OUTPUTF = "data" + File.separator +
"sunsetcopyf.jpg";

    public static void copyBinary() throws FileNotFoundException,
IOException {
        FileInputStream in = new FileInputStream(INPUT);
        FileOutputStream out = new FileOutputStream(OUTPUT);
        int data;
        while (true) {
            data = in.read();
            if (data == -1) {
                break;
            } else {
                out.write(data);
            }
        }
        in.close();
        out.close();
    }

    public static void copyBinaryFast() throws FileNotFoundException,
IOException {
        BufferedInputStream in = new BufferedInputStream(new
FileInputStream(INPUT));
        BufferedOutputStream out = new BufferedOutputStream(new
FileOutputStream(OUTPUTF));
        int data;
        while (true) {
            data = in.read();
            if (data == -1) {
                break;
            } else {
                out.write(data);
            }
        }
        in.close();
```

```
            out.close();
        }

        public static void main(String[] args) throws IOException {
            copyBinary();
            copyBinaryFast();
        }
    }
```

Κώδικας 15.5 *Ανάγνωση και εγγραφή σε αρχεία ψηφιολέξεων*

Η κλάση BinaryData περιλαμβάνει δύο στατικές συναρτήσεις, την copyBinary και την copyBinaryFast. Και οι δύο αντιγράφουν την εικόνα sunset.jpg σε νέο αρχείο. Η copyBinary χρησιμοποιεί ένα αντικείμενο FileInputStream για ανάγνωση ψηφιολέξεων και ένα αντικείμενο FileOutputStream για εγγραφή ψηφιολέξεων. Η copyBinaryFast χρησιμοποιεί επιπλέον τα περιβλήματα BufferedInputStream και BufferedOutputStream. Δεδομένου ότι και οι δύο επιτελούν την ίδια εργασία, είναι εύκολη η εκτίμηση της επιτάχυνσης που προσφέρουν οι ροές προσωρινής μνήμης στις εφαρμογές. Τρέξτε τις δύο συναρτήσεις χωριστά και εκτιμήστε τη χρονική διαφορά στην εκτέλεσή τους.

Μετά το άνοιγμα της ροής, μέσα στην επαναληπτική διαδικασία, οι συναρτήσεις διαβάζουν μια προς μια τις ψηφιολέξεις. Η read διαβάζει την τρέχουσα ψηφιολέξη την οποία και επιστρέφει, οπότε στη συνέχεια η write την γράφει στο αρχείο εξόδου. Αν ωστόσο η read συναντήσει τέλος αρχείου, τότε επιστρέφει -1, οπότε διακόπτεται η επαναληπτική διαδικασία. Τέλος, οι συναρτήσεις κλείνουν τις ροές.

15.4 Αρχεία τυχαίας προσπέλασης

Ένας άλλος πολύ σημαντικός τρόπος διαχείρισης των αρχείων είναι η διαχείριση με τυχαία προσπέλαση (Random access). Η τυχαία προσπέλαση μας δίνει τη δυνατότητα να διαβάσουμε ή να γράψουμε άμεσα σε οποιοδήποτε σημείο του αρχείου θέλουμε. Ο χρόνος που απαιτείται για την προσπέλαση είναι ίδιος είτε προσπελαύνουμε την αρχή του αρχείου είτε οποιοδήποτε άλλο σημείο.

Η τυχαία προσπέλαση αξιοποιείται συνήθως σε δομημένα αρχεία. Ας πάρουμε για παράδειγμα το αρχείο student.dat που δημιουργήθηκε από τον κώδικα 15.4. Σε αυτό τα στοιχεία κάθε μαθητή καταλαμβάνουν μία γραμμή και συνιστούν μια εγγραφή (record). Όλες οι εγγραφές στο αρχείο έχουν το ίδιο πλάτος. Καθώς η συνάρτηση που παράγει αυτές τις εγγραφές, η toFixedWidth βρίσκεται στην κλάση Student, εκεί έχουμε τοποθετήσει και την πληροφορία σχετικά με το μήκος της εγγραφής. Πρόκειται για τη στατική μεταβλητή RECSIZE. Τα αρχεία τυχαίας προσπέλασης υποστηρίζουν τον δείκτη αρχείου (file pointer). Οι λειτουργίες ανάγνωσης και εγγραφής αρχίζουν από το σημείο που υποδεικνύει ο δείκτης αρχείου. Αν για παράδειγμα, θέλουμε να διαβάσουμε την πρώτη εγγραφή του αρχείου, ο δείκτης πρέπει να δείχνει στην ψηφιολέξη 0, αν θέλουμε να διαβάσουμε τη δεύτερη εγγραφή, ο δείκτης πρέπει να δείχνει στην ψηφιολέξη RECSIZE, αν θέλουμε την τρίτη εγγραφή, ο δείκτης πρέπει να δείχνει στην ψηφιολέξη 2*RECSIZE, κοκ.

Η κλάση που μας εξυπηρετεί για τυχαία προσπέλαση είναι η RandomAccessFile. Ο δείκτης του αρχείου τοποθετείται με τη μέδοθο seek. Επίσης, μπορούμε να μάθουμε πού δείχνει ο δείκτης με τη μέθοδο getFilePointer.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.time.LocalDate;
import java.time.Month;

public class RandomAccess {

    public static Student read(RandomAccessFile f, int recNo) throws
IOException {
        int records = (int) (f.length() / Student.RECSIZE);
        if (recNo < 0 || recNo >= records) {
            throw new RuntimeException("reading past of file end");
```

```
            }
            long pos = recNo * Student.RECSIZE;
            f.seek(pos);
            String s = f.readLine();
            Student student = StudentsToStdWidth.toStudent(s);
            return student;
        }

        public static void update(RandomAccessFile f, int recNo, Student student)
    throws IOException {
            int records = (int) (f.length() / Student.RECSIZE);
            if (recNo < 0 || recNo >= records) {
                throw new RuntimeException("updating past of file end");
            }
            long pos = recNo * Student.RECSIZE;
            f.seek(pos);
            f.writeBytes(student.toFixedWidth());
        }

        public static void append(RandomAccessFile f, Student student) throws
    IOException {
            f.seek(f.length());
            f.writeBytes(student.toFixedWidth()+"\n");
        }

        public static void main(String[] args) throws FileNotFoundException,
    IOException {
            RandomAccessFile f = new
    RandomAccessFile("data"+File.separator+"students.dat", "rw");
            Student student = read(f, 3);
            System.out.println(student);
            student.setfName("Mike");
            student.setsName("Updater");
            update(f, 3, student);
            student=new Student("999", "Jane", "Appender",
    LocalDate.of(1999,Month.MARCH,20));
            student.setScores(new double[]{14,15,16,17,18,19});
            append(f,student);
            append(f,student);
            f.close();
        }
    }
```

Κώδικας 15.6 *Ανάγνωση και εγγραφή σε αρχεία με τυχαία προσπέλαση*

Η συνάρτηση read στον κώδικα 15.6 λαμβάνει ως παραμέτρους ένα αντικείμενο RandomAccessFile και έναν ακέραιο. Ο ακέραιος αναπαριστά τον αύξοντα αριθμό της εγγραφής μέσα στο αρχείο. Η συνάρτηση length της RandomAccessFile επιστρέφει το συνολικό μήκος του αρχείου σε ψηφιολέξεις. Το μήκος αυτό διαιρούμενο δια του μήκους της εγγραφής μας δίνει τον αριθμό των εγγραφών στο αρχείο. Έτσι, γίνεται έλεγχος μήπως ο αριθμός εγγραφής που ζητήθηκε από την read είναι εκτός των ορίων του αρχείου. Αν όχι, ο δείκτης τοποθετείται στη θέση recNo * Student.RECSIZE. Μετά την τοποθέτηση του δείκτη, διαβάζουμε μια εγγραφή και δημιουργούμε το αντικείμενο Student με την toStudent της StudentsToStdWidth.

Η συνάρτηση append μετακινεί τον δείκτη στο τέλος του αρχείου όπου προσθέτει μια νέα εγγραφή με τη βοήθεια της writeBytes και της toFixedWidth.

Η συνάρτηση update λαμβάνει ως παραμέτρους το αρχείο, τον αριθμό εγγραφής και ένα αντικείμενο Student. Αντικαθιστά τα στοιχεία της εγγραφής που βρίσκεται στη θέση recNo*Student.RECSIZE με τα στοιχεία του αντικειμένου Student. Προσέξτε πως στην writeBytes δεν προστίθεται χαρακτήρας newline όπως στην αντίστοιχη writeBytes της append, καθώς ο χαρακτήρας υπάρχει ήδη στο αρχείο.

15.5 Σειριοποίηση

Ας υποθέσουμε πως σε μια εφαρμογή-παιχνίδι θέλουμε να δώσουμε τη δυνατότητα στον χρήστη να διακόπτει το παιχνίδι και να το συνεχίζει ακριβώς από το σημείο στο οποίο διέκοψε. Στην περίπτωση αυτή θα μας ήταν χρήσιμο να αποθηκεύσουμε σε αρχείο τα αντικείμενα της εφαρμογής όπως αυτά είχαν διαμορφωθεί στη μνήμη κατά τη στιγμή της διακοπής έτσι ώστε να μπορούμε να τα φορτώσουμε από το αρχείο στη μνήμη όποτε χρειαστεί. Η διαδικασία με την οποία επιτυγχάνουμε αποθήκευση και φόρτωση αντικειμένων μεταξύ μνήμης και αρχείου ονομάζεται σειριοποίηση (serialization).

Θα δούμε τη σειριοποίηση στην πράξη με ένα παράδειγμα το οποίο σώζει ένα ArrayList από αντικείμενα κλάσης που αναπαριστά αυτοκίνητα και στη συνέχεια το φορτώνει και το εμφανίζει. Θα χρειαστούμε καταρχάς μια κλάση που αναπαριστά τα αυτοκίνητα.

```
import java.io.Serializable;
import java.util.Objects;

public class Car implements Serializable {
    private String brand;
    private int maximumSpeed;

    public Car(String brand, int maximumSpeed) {
        this.brand = brand;
        this.maximumSpeed = maximumSpeed;
    }

    public int getMaximumSpeed() {
        return maximumSpeed;
    }

    public void setMaximumSpeed(int maximumSpeed) {
        this.maximumSpeed = maximumSpeed;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    @Override
    public String toString() {
        return "Car{" + "brand=" + brand + ", maximumSpeed=" + maximumSpeed
+ '}';
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 67 * hash + Objects.hashCode(this.brand);
        hash = 67 * hash + this.maximumSpeed;
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
```

```
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Car other = (Car) obj;
        if (this.maximumSpeed != other.maximumSpeed) {
            return false;
        }
        return Objects.equals(this.brand, other.brand);
    }
}
```

Κώδικας 15.7 *Η κλάση Car*

Η κλάση Car στον κώδικα 15.7 είναι μια απλή κλάση που περιλαμβάνει τις αναγκαίες συναρτήσεις. Εκείνο που έχει παραπάνω σε σχέση με τις κλάσεις που έχουμε υλοποιήσει μέχρι τώρα είναι πως υλοποιεί την προκαθορισμένη διεπαφή Serializable. Η διεπαφή Serializable δεν ορίζει καμία αφηρημένη μέθοδο. Πρόκειται για διεπαφή-δείκτη (marker interface). Η υλοποίησή της από οποιαδήποτε κλάση απαιτεί μόνο τη δήλωσή της στη λίστα των διεπαφών που υλοποιεί η κλάση. Ωστόσο είναι απαραίτητη ώστε να επιτρέπεται η σειριοποίηση μιας κλάσης.

Ας προχωρήσουμε τώρα στην εφαρμογή που αποθηκεύει το ArrayList από αντικείμενα Car.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

public class Serialization {

    static ArrayList<Car> createCars() {
        ArrayList<Car> store = new ArrayList<>();
        store.add(new Car("Toyota", 220));
        store.add(new Car("Ford", 230));
        store.add(new Car("Peugeot", 240));
        return store;
    }

    static void saveCars() throws IOException {
        ArrayList<Car> cars = createCars();
        ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data\\cars.dat"));
        out.writeObject(cars);
        out.close();
    }

    static ArrayList<Car> loadCars() throws IOException,
ClassNotFoundException {
        ObjectInputStream in = new ObjectInputStream(
                new FileInputStream("data\\cars.dat"));
        ArrayList<Car> loaded = (ArrayList<Car>) in.readObject();//essential
unsafe operation
        in.close();
        return loaded;
    }

    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        saveCars();
```

```
            ArrayList<Car> store = loadCars();
            System.out.println(store);
        }
    }
```

Κώδικας 15.8 *Σειριοποίηση ενός ArrayList<Car>*

Στον κώδικα 15.8 η createCars δημιουργεί το ArrayList από αντικείμενα Car. Η saveCars ανοίγει ροή τύπου ObjectOutputStream που λειτουργεί ως περίβλημα σε FileOutputStream. Με την writeObject πραγματοποιείται η εγγραφή. Προσέξτε στην τεκμηρίωση της ArrayList πως η κλάση υλοποιεί τη διεπαφή Serializable, διαφορετικά η λειτουργία θα αποτύγχανε. Αντίστοιχα απλή είναι και η διαδικασία ανάγνωσης του αντικειμένου που γίνεται από την loadCars με χρήση ενός ObjectInputStream και ενός FileInputStream. Προσέξτε πως η readObject της ObjectInputStream επιστρέφει Object και άρα είναι απαραίτητη η μετατροπή του σε ArrayList<Car>.

15.6 Χρήσιμες λειτουργίες

Η κλάση File παρέχει μια σειρά από χρήσιμες λειτουργίες. Ένα αντικείμενο της κλάσης File αναπαριστά το όνομα της διαδρομής ενός αρχείου ή φακέλου [2]. Τα λειτουργικά συστήματα ονομάζουν τις διαδρομές των αρχείων με βάση τα ιδιαίτερα χαρακτηριστικά τους. Αυτά όμως διαφέρουν από λειτουργικό σε λειτουργικό. Καθώς η Java είναι διαπλατφορμική γλώσσα, χρειάζεται έναν τρόπο για να αναπαριστά τις ονομασίες των διαδρομών αρχείων και φακέλων που είναι ανεξάρτητος από το λειτουργικό σύστημα. Αυτόν τον τρόπο παρέχει η κλάση File.

Οι ονομασίες των διαδρομών αρχείων και φακέλων είναι δύο τύπων, οι σχετικές (relative) και οι πλήρεις (absolute). Η πλήρης διαδρομή επαρκεί για τον εντοπισμό του αρχείου που δηλώνει. Αντίθετα, μια σχετική διαδρομή πρέπει να συνδυαστεί με κάποια άλλη προκειμένου να εντοπιστεί το αρχείο που δηλώνει. Οι κλάσεις του πακέτου java.io επιλύουν τις σχετικές διαδρομές με βάση την ιδιότητα user.dir του συστήματος. Στην περίπτωσή μας, εφόσον δουλεύουμε στο NetBean, η διαδρομή που συνδυάζεται με την ιδιότητα user.dir αντιστοιχεί στον φάκελο του NetBeans project στο οποίο εργαζόμαστε.

Ας δούμε τώρα μια σειρά από χρήσιμες συναρτήσεις:

```
public static String getProperty(String key)
```

Η συνάρτηση getProperty της κλάσης System όταν κληθεί με παράμετρο “user.dir” επιστρέφει τον τρέχοντα φάκελο εργασίας.
Οι υπόλοιπες συναρτήσεις ανήκουν στην κλάση File.

```
boolean canExecute()
```

Ελέγχει αν η εφαρμογή μπορεί να τρέξει αυτό το αρχείο, δηλαδή το αρχείο που αναπαρίσταται από το αντικείμενο File που καλεί την canExecute.

```
boolean canRead()
```

Ελέγχει αν η εφαρμογή μπορεί να διαβάσει αυτό το αρχείο.

```
boolean canWrite()
```

Ελέγχει αν η εφαρμογή μπορεί να γράψει σε αυτό το αρχείο

```
boolean createNewFile()
```

Δημιουργεί ένα αρχείο με αυτό το όνομα εφόσον δεν υπάρχει ήδη.

```
boolean delete()
```

Διαγράφει το αρχείο ή τον φάκελο με αυτό το όνομα.

```
String getAbsolutePath()
```

Επιστρέφει την πλήρη διαδρομή αυτού του αρχείου.

```
String getName()
```

Επιστρέφει το όνομα αυτού του αρχείου.

```
boolean isDirectory()
```

Ελέγχει αν αυτό το αρχείο είναι φάκελος.

```
boolean isHidden()
```

Ελέγχει αν αυτό το αρχείο είναι κρυφό.

```
long lastModified()
```

Επιστρέφει τον χρόνο σε milisecond από την τελευταία τροποποίηση αυτού του αρχείου.

```
long length()
```

Επιστρέφει το μέγεθος σε bytes αυτού του αρχείου.

```
File[] listFiles()
```

Επιστρέφει έναν πίνακα από αρχεία που περιέχονται σε αυτόν τον φάκελο.

```
boolean mkdir()
```

Δημιουργεί έναν φάκελο με αυτό το όνομα.

15.7 Λυμένες Ασκήσεις

15.7.1 Περιοδικός Πίνακας

Σε ένα οριοθετημένο αρχείο κειμένου περιέχονται **όλα** τα χημικά στοιχεία του περιοδικού πίνακα με τα βασικά τους χαρακτηριστικά όπως φαίνεται στο ακόλουθο παράδειγμα:

```
H 1 1 s 1
Li 1 2 s 3
Na 1 3 s 11
Be 2 2 s 4
Mg 2 3 s 12
Sc 3 4 d 21
Y 3 5 d 39
He 18 1 p 2
La 3 6 f 57
```

Η γραμμογράφηση του αρχείου έχει ως εξής:

```
Σύμβολο Ομάδα Περίοδος Τομέας Ατομικός_Αριθμός
```

Προσθέστε στην κλάση TableOfChemichalElements συνάρτηση

```
public static TableOfChemichalElements getPeriodTable()
```

που επιστρέφει ένα αντικείμενο TableOfChemichalElements που περιλαμβάνει τα στοιχεία του αρχείου και επομένως αναπαριστά τον περιοδικό πίνακα. Φροντίστε ώστε η getPeriodTable να διαβάζει το αρχείο μόνο μία φορά κατά την τρέχουσα εκτέλεση του προγράμματος. Μετά την ανάγνωση να διατηρεί την επιστρεφόμενη τιμή στη μνήμη από όπου την επιστρέφει με κάθε κλήση της.

Λύση
Καταρχάς θα πρέπει να προσθέσουμε δύο στατικά μέλη στην κλάση.

```
private final static File F = new File("data" +
File.separator + "ChemicalElements.dat");
private static TableOfChemichalElements periodTable = null;
```

Το F αφορά το αρχείο το οποίο περιέχει τα χημικά στοιχεία. Η getPeriodTable δεν θα φορτώνει δεδομένα από ένα οποιοδήποτε αρχείο αλλά από ένα συγκεκριμένο που περιέχει το σύνολο των χημικών στοιχείων. Με αυτήν την έννοια, το αρχείο δεν σχετίζεται με συγκεκριμένο στιγμιότυπο της TableOfChemichalElements, οπότε το F είναι static. Επίσης static είναι το periodTable που είναι ο πίνακας που επιστρέφει η getPeriodTable. Το periodTable αρχικοποιείται στην τιμή null. Μετά την πρώτη εκτέλεση της getPeriodTable αποτελεί αναφορά στο στιγμιότυπο TableOfChemichalElements που επιστρέφει η getPeriodTable.
Στη συνέχεια, αναπτύσσουμε τρεις εναλλακτικές λύσεις και συζητάμε τα πλεονεκτήματα και μειονεκτήματά τους.

```
public static TableOfChemichalElements getPeriodTable() throws
FileNotFoundException {
        if (periodTable == null) {
            periodTable = new TableOfChemichalElements();
            Scanner scanner = new Scanner(F);
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                String[] elements = line.split(" ");
                ChemicalElement cE = new
ChemicalElement(Integer.parseInt(elements[1]), Integer.parseInt(elements[2]),
elements[3].charAt(0), Integer.parseInt(elements[4]), elements[0]);
                periodTable.add(cE);
            }
            scanner.close();
        }
        return periodTable;
    }
```

Κώδικας 15.9 *Η συνάρτηση getPeriodTable*

Στον κώδικα 15.9 η συνάρτηση getPeriodTable ελέγχει τη στατική μεταβλητή periodTable και αν την βρει null φορτώνει το αρχείο με τα χημικά στοιχεία στη μνήμη. Στη συνέχεια επιστρέφει την periodTable. Πράγματι με αυτήν την προσέγγιση θα επιστρέψει στον χρήστη της TableOfChemichalElements έναν πίνακα με χημικά στοιχεία. Έχει ενδιαφέρον όμως τι μπορεί να συμβεί από κει και πέρα.

```
public static void main(String[] args) throws IOException {
   TableOfChemichalElements pT=TableOfChemichalElements.getPeriodTable();
   ChemicalElement li = new ChemicalElement(1, 2, 's', 3, "Pi");
   pT.add(li);
   System.out.println(TableOfChemichalElements.getPeriodTable());

}
```

Κώδικας 15.10 *Αλλοίωση του πίνακα περιοδικών στοιχείων από την εφαρμογή*

Η getPeriodTable επιστρέφει την αναφορά στον TableOfChemicalElements.periodTable. Ο κώδικας 15.10 κρατάει αυτήν την αναφορά στην τοπική μεταβλητή pT. Επομένως, οι αναφορές TableOfChemicalElements.periodTable και pT βλέπουν στον ίδιο πίνακα στη μνήμη. Από τη στιγμή που θα γίνει μια μεταβολή στοιχείων του πίνακα από την εφαρμογή, π.χ. στον κώδικα 15.10 προστίθεται ένα στοιχείο, κάθε κλήση της getPeriodTable θα επιστρέφει όχι τον αρχικό πίνακα, αυτόν που φορτώθηκε από το αρχείο, αλλά τον πίνακα όπως μεταβλήθηκε από την εφαρμογή. Πρόκειται όμως για τον περιοδικό πίνακα των χημικών στοιχείων που είναι συγκεκριμένος και θα πρέπει πάντα να επιστρέφεται αμετάβλητος. Επομένως, η προσέγγιση αυτή παρουσιάζει σοβαρή αδυναμία.

Η λύση στο πρόβλημα αυτό είναι η getPeriodTable να δημιουργεί έναν νέο πίνακα, να τον ενημερώνει κατάλληλα και να τον επιστρέφει στον χρήστη της. Έτσι κάθε μεταβολή του πίνακα από την εφαρμογή δεν θα

επηρεάζει τον πίνακα που θα επιστρέφει η getPeriodTable. Για να το πετύχουμε αυτό, θα πρέπει καταρχάς να προσθέσουμε δύο νέους δημιουργούς στην TableOfChemicalElements.

```
public TableOfChemichalElements() {}

public TableOfChemichalElements(TableOfChemichalElements source) {
    Iterator<ChemicalElement> it = source.store.iterator();
    while (it.hasNext()) {
        store.add(it.next());
    }
}
```

Κώδικας 15.11 *Δύο δημιουργοί της κλάσης TableOfChemicalElements.*

Ο δημιουργός που λαμβάνει ως παράμετρο TableOfChemicalElements παράγει ένα αντίγραφο της παραμέτρου του. Προσέξτε πως τα χημικά στοιχεία στο αντίγραφο είναι τα ίδια με τα χημικά στοιχεία της παραμέτρου και όχι αντίγραφά τους. Αυτό είναι σωστό καθώς τα χημικά στοιχεία δεν διαθέτουν ρυθμιστές και επιπλέον όλες οι μεταβλητές στιγμιότυπου είναι σταθερές. Επομένως, ένα χημικό στοιχείο από τη στιγμή που θα δημιουργηθεί δεν μπορεί να μεταβληθεί και άρα δεν υπάρχει λόγος να παράγουμε αντίγραφο.

Ο δημιουργός χωρίς παραμέτρους είναι απαραίτητος για να τρέχουν οι κώδικες που μέχρι τώρα χρησιμοποιούσαν τον εξ ορισμού δημιουργό.

Έχοντας τους δύο δημιουργούς στη διάθεσή μας αρκεί να μετατρέψουμε την τελευταία γραμμή της getPeriodTable ώστε, αντί να επιστρέφει τη στατική periodTable, να επιστρέφει ένα αντίγραφό της όπως δείχνει ο κώδικας που ακολουθεί:

```
return new TableOfChemichalElements(periodTable);
```

Μία εναλακτική προσέγγιση βασίζεται στην αξιοποίηση της διεπαφής Cloneable. Όσες κλάσεις υλοποιούν τη διεπαφή Cloneable τους επιτρέπεται να καλέσουν τη συνάρτηση clone() που κληρονομούν από την Object. Εδώ το σχήμα φαίνεται λίγο περίεργο καταρχάς, αλλά έχει την εξήγησή του. Η συνάρτηση clone() που επιστρέφει ένα αντίγραφο του αντικειμένου που την καλεί τοποθετήθηκε στην κλάση Object σε αρχική έκδοση της Java. Επομένως, όλες οι κλάσεις την κληρονομούν. Σύντομα όμως έγινε αντιληπτό πως κάποιες κλάσεις δεν θα πρέπει να δίνουν τη δυνατότητα παραγωγής αντιγράφων των αντικειμένων τους. Προκειμένου να μην βγει η clone από την Object ώστε να διατηρηθεί η συμβατότητα με προηγούμενες εκδόσεις, οι σχεδιαστές της Java αποφάσισαν να ορίσουν τη διεπαφή Cloneable και να επιτρέψουν τη χρήση της clone στις κλάσεις που υλοποιούν την Cloneable. Επομένως, ο λόγος της περίεργης απαίτησης υλοποίησης μιας διεπαφής προκειμένου να υλοποιηθεί μια μέθοδος που κληρονομείται κρύβεται πίσω από μια αρχικά λανθασμένη σχεδιαστική επιλογή και μια εκ των υστέρων ευρετική λύση. Ας σημειωθεί πως οι κλάσεις που επιχειρούν να καλέσουν την clone χωρίς να έχουν υλοποιήσει τη διεπαφή clone παράγουν εξαίρεση τύπου CloneNotSupportedException.

Για την υλοποίηση αυτής της εναλακτικής θα πρέπει καταρχάς να δηλώσουμε στην επικεφαλίδα της TableOfChemicalElements ότι υλοποιεί τη διεπαφή Cloneable, όπως φαίνεται αμέσως παρακάτω:

```
public class TableOfChemichalElements implements Cloneable {…}
```

Στη συνέχεια πρέπει να επανορίσουμε τη συνάρτηση clone στην TableOfChemichalElements.

```
@Override
public Object clone() throws CloneNotSupportedException {
    TableOfChemichalElements rVal=new TableOfChemichalElements();
    Iterator<ChemicalElement> it = store.iterator();
    while (it.hasNext()) {
        rVal.add(it.next());
    }
    return rVal;
}
```

Κώδικας 15.12 *Η συνάρτηση clone της κλάσης TableOfChemicalElements.*

Προσέξτε πως η clone παράγει CloneNotSupportedException και επιστρέφει Object. Στη συνέχεια θα τροποποιήσουμε την getPeriodTable ως εξής:

```
    public static TableOfChemichalElements getPeriodTable() throws
FileNotFoundException, CloneNotSupportedException {
        if (periodTable == null) {
            periodTable = new TableOfChemichalElements();
            Scanner scanner = new Scanner(F);
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                String[] elements = line.split(" ");
                ChemicalElement cE = new
ChemicalElement(Integer.parseInt(elements[1]), Integer.parseInt(elements[2]),
elements[3].charAt(0),
                        Integer.parseInt(elements[4]), elements[0]);
                periodTable.add(cE);
            }
            scanner.close();
        }
        return (TableOfChemichalElements)periodTable.clone();
    }
```

Κώδικας 15.13 *Η συνάρτηση getPeriodTable βασισμένη στην clone.*

Στον κώδικα 15.13 η επιστροφή της συνάρτησης βασίζεται στην clone. Καθώς η clone επιστρέφει αντικείμενο τύπου Object, πρέπει υποχρεωτικά να προβούμε σε μετατροπή τύπου. Επιπλέον, στη συνάρτηση έχει δηλωθεί η πιθανή παραγωγή της CloneNotSupportedException.

Ανακεφαλαιώνοντας επισημαίνουμε πως η πρώτη λύση παρουσιάζει σοβαρή αδυναμία που μπορεί να οδηγήσει σε λάθη τον κώδικα μιας εφαρμογής. Αντίθετα, η δεύτερη και τρίτη λύση προστατεύει τον περιοδικό πίνακα. Οι χρήστες της κλάσης μπορούν να κάνουν μεταβολές στον πίνακα που θα λάβουν από την getPeriodTable αλλά η τιμή επιστροφής της μένει αμετάβλητη. Στην πράξη η δεύτερη και η τρίτη λύση είναι σε γενικές γραμμές ισοδύναμες.

15.7.2 Κατάλογοι αρχείων

Να υλοποιηθεί συνάρτηση public static void dirRecurse(File root) που λαμβάνει ως παράμετρο και εμφανίζει τα περιεχόμενα του root. Αν το root αντιστοιχεί σε κατάλογο αρχείων, εμφανίζει στοιχεία για κάθε αρχείο του καταλόγου και το ίδιο κάνει για κάθε υποκατάλογο του root. Αν το root αντιστοιχεί σε απλό αρχείο, εμφανίζει πληροφορίες για το αρχείο. Για κάθε κατάλογο από το root και κάτω θα πρέπει να εμφανίζεται πρώτα η ονομασία του και να ακολουθεί λίστα με πληροφορίες για κάθε αρχείο του καταλόγου σύμφωνα με το υπόδειγμα που ακολουθεί:

```
   Directory: G:\My Drive\Code\Kallipos\book\data
-rw     414 B      students.dat
-rw     220 B      studentsd.dat
-rw     75 KB      sunset.jpg
-rw     75 KB      sunsetcopy.jpg
-rw     75 KB      sunsetcopyf.jpg
-rw     199 B      cars.dat
-rw     122 B      doubledata.dat
-rw     103 B      ChemicalElements.dat

   Directory: G:\My Drive\Code\Kallipos\book\data\test
-rw      72 B      Calendar.txt
-rw     16 KB     Letter.docx
hrw       0 B      demohidden.txt
```

Στο υπόδειγμα αυτό η dirRecurse κλήθηκε με παράμετρο το αντικείμενο File που αντιστοιχεί στον φάκελο G:\My Drive\Code\Kallipos\book\data. Ο φάκελος περιέχει τον υποφάκελο test. Οι πληροφορίες που αφορούν κάθε αρχείο έχουν ως εξής: Οι πρώτοι τρεις χαρακτήρες δηλώνουν κατά σειρά αν το αρχείο είναι κρυφό (hidden), αν επιτρέπεται η ανάγνωση και η εγγραφή. Στη συνέχεια ακολουθεί το μέγεθος του αρχείου και τέλος το όνομά του.

Λύση
Καταρχάς θα συγκεντρώσουμε όλες τις συναρτήσεις που θα μας χρειαστούν για να δομήσουμε την dirRecurse σε μία κλάση την οποία ονομάζουμε Directory.

```
import java.io.File;
import java.util.Locale;

public class Directory {

    public static String convertBytes(long bytes) {
        String rVal;
        long kB = bytes / 1024;
        long mB = bytes / (long) Math.pow(1024, 2);
        long gB = bytes / (long) Math.pow(1024, 3);
        if (gB > 0) {
            rVal = gB + " GB";
        } else if (mB > 0) {
            rVal = mB + " MB";
        } else if (kB > 0) {
            rVal = kB + " KB";
        } else {
            rVal = bytes + " B";
        }
        return rVal;
    }

    public static void dirRecurse(File root) {
        if (root.isDirectory()) {
            System.out.println("   Directory: " + root.getAbsolutePath());
            showFiles(root.listFiles());
        } else {
            System.out.println((root.isHidden() ? "h" : "-")
                    + (root.canRead() ? "r" : "-")
                    + (root.canWrite() ? "w" : "-")
                    + String.format(Locale.US, "%10s",
convertBytes(root.length()))
                    + "      " + root.getName());
        }
    }

    public static void showFiles(File[] files) {
        for (File file : files) {
            if (file.isDirectory()) {
                System.out.println("");
                System.out.println("   Directory: " +
file.getAbsolutePath());
                showFiles(file.listFiles()); // Calls same method again.
            } else {
                System.out.println((file.isHidden() ? "h" : "-")
                        + (file.canRead() ? "r" : "-")
                        + (file.canWrite() ? "w" : "-")
                        + String.format(Locale.US, "%10s",
convertBytes(file.length()))
                        + "      " + file.getName());
```

```
                }
            }
        }

        public static void main(String[] args) {
            File root = new File("data");
            dirRecurse(root);
        }
    }
```

Κώδικας 15.14 *Η κλάση Directory.*

Η πιο κρίσιμη ίσως συνάρτηση της Directory στον κώδικα 15.14 είναι η showFiles. Η showFiles λαμβάνει ως παράμετρο έναν πίνακα από αντικείμενα Files και για κάθε ένα από αυτά ελέγχει και αν είναι φάκελος, τυπώνει τη διαδρομή του και προχωράει σε αναδρομική κλήση με τη λίστα που αναπαριστά τα αρχεία του τρέχοντος φακέλου. Αν όμως πρόκειται για απλό αρχείο, τότε απλώς εμφανίζει τις απαιτούμενες πληροφορίες. Οι πληροφορίες παράγονται με αξιοποίηση των κατάλληλων συναρτήσεων της κλάσης File. Ειδικά για το μέγεθος του αρχείου χρησιμοποιείται και η συνάρτηση convertBytes. Καθώς η συνάρτηση length επιστρέφει το μέγεθος του αρχείου σε bytes, υπάρχει η ανάγκη να μετατραπεί σε KB, MB ή GB ανάλογα με τον αριθμό των bytes. Τη μετατροπή αυτή αναλαμβάνει η convertBytes, η οποία διαιρεί τον αριθμό των bytes με το 1024, το 1024^2 και το 1024^3 ώστε να βρει την κατάλληλη έκφραση για το μέγεθος του αρχείου. Αν η διαίρεση δώσει θετικό αριθμό GB, τότε το μέγεθος θα εκφραστεί σε GB. Αν όμως δεν προκύψει από τη διαίρεση θετικός αριθμός GB, περίπτωση κατά την οποία το μέγεθος του αρχείου είναι μικρότερο από 1 GB, τότε εξετάζεται η περίπτωση που το μέγεθος του αρχείου είναι μεγαλύτερο από 1 MB ή 1KB.

Τέλος η dirRecurse αποτελεί περίβλημα της showFiles. Η dirRecurse λαμβάνει ως παράμετρο ένα αντικείμενο File. Αν είναι φάκελος, τυπώνει την ονομασία του και καλεί την showFiles με τη λίστα αρχείων του φακέλου. Αν όμως η παράμετρος της dirRecurse αντιστοιχεί σε αρχείο, εμφανίζει απλώς τα στοιχεία του.

15.8 Ασκήσεις προς Λύση

15.8.1 Quiz 2η έκδοση

Να τροποποιηθεί η κλάση Quiz έτσι ώστε οι ερωτήσεις να φορτώνονται από αρχείο. Η κλάση θα πρέπει να συνεχίσει να υποστηρίζει ερωτήσεις σωστού λάθους και πολλαπλής επιλογής. Το αρχείο στο οποίο βρίσκονται οι ερωτήσεις να δίνεται ως παράμετρος της main.

15.8.2 Grep

Οι εντολές grep (global regular expression print) ψάχνουν σε μία ομάδα αρχείων για να βρουν μια συμβολοσειρά. Σε αυτήν την άσκηση θα πρέπει να αναπτύξουμε μια συνάρτηση public static void grep(File file, String searchItem) που λαμβάνει δύο παραμέτρους. Αν η πρώτη παράμετρος αντιστοιχεί σε απλό αρχείο, ψάχνει στο αρχείο να βρει τη συμβολοσειρά searchItem και εμφανίζει κάθε γραμμή του αρχείου που περιέχει την searchItem μαζί με τον αριθμό που έχει η γραμμή μέσα στο αρχείο. Αν η πρώτη παράμετρος είναι φάκελος, εφαρμόζει την αναζήτηση που περιγράφηκε για κάθε αρχείο του φακέλου και για τα αρχεία όλων των υποφακέλων του. Ακολουθήστε το παρακάτω υπόδειγμα εξόδου της grep:

students.dat

```
(1)  113            Jim          Blue10.011.012.013.014.015.02005-03-01
(5)  999            Jim       Appender14.015.016.017.018.019.01999-03-20
```

studentsd.dat

```
(1) 113 Jim Blue 10.0 11.0 12.0 13.0 14.0 15.0 2005-04-01
```

Ο αριθμός μέσα στις παρενθέσεις αναπαριστά τον αριθμό γραμμής του αρχείου όπου βρέθηκε το searchItem. Στο συγκεκριμένο παράδειγμα το searchItem είναι η σειρά Jim.

15.8.3 CopyFile

Αναπτύξτε java εφαρμογή που λαμβάνει ως παραμέτρους δύο συμβολοσειρές source και target και αντιγράφει το αρχείο με διαδρομή το source στο αρχείο target. Οι παράμετροι source και target θα πρέπει να περνούν στην main. Ελέγξτε αν η εφαρμογή σας λειτουργεί σωστά τόσο με αρχεία κειμένου όσο και με αρχεία εικόνας.

Βιβλιογραφία

[1] C. Horstmann, Η γλώσσα προγραμματισμού Java, Αναλυτική Προσέγγιση. Broken Hill Publishers, 2021.

[2] J. Farrell, Java, Εκμάθηση με πρακτικά παραδείγματα. Κριτική, 2018.

Κεφάλαιο 16

Σύνοψη
Στην ενότητα αυτή παρουσιάζεται ο μηχανισμός των εξαιρέσεων. Καταρχάς εξηγείται η λειτουργία της στοίβας κλήσης, στη συνέχεια διαφοροποιούνται οι ελεγχόμενες από τις προαιρετικά ελεγχόμενες εξαιρέσεις και τα Errors. Μετά, παρουσιάζονται κατά σειρά, οι τεχνικές σύλληψης, η πρόταση finally, οι εξαιρέσεις που ορίζονται από τον χρήστη, ο μηχανισμός διαχείρισης και η σύνθετη σύλληψη. Τέλος, παρουσιάζονται οι βασικές προκαθορισμένες εξαιρέσεις της Java.

Προαπαιτούμενη γνώση
Η ενσωμάτωση, η κληρονομικότητα και ο πολυμορφισμός στο αντικειμενοστρεφές μοντέλο με Java, οι αφηρημένες κλάσεις καθώς και οι βασικές προκαθορισμένες κλάσεις, δηλαδή οι String, StringBuilder, ArrayList, LinkedList, Stack, PriorityQueue, HashSet, HashMap, HashTree, LocalDate, Localtime, Period και Duration. Οι διεπαφές στην Java. Η βασική διαχείριση των αρχείων.

Λέξεις κλειδιά
Στοίβα κλήσης (call stack), Εξαίρεση (exception), Error, ελεγχόμενη (checked) εξαίρεση, προαιρετικά ελεγχόμενη (unchecked) εξαίρεση.

16 Εξαιρέσεις

Μερικές φορές κατά την εκτέλεση μιας εφαρμογής είναι δυνατόν να συμβεί κάποιο απροσδόκητο γεγονός τέτοιο που η εφαρμογή δεν μπορεί να διαχειριστεί τουλάχιστον στο πλαίσιο της κανονικής ροής της. Για παράδειγμα, η εφαρμογή χρειάζεται να διαβάσει δεδομένα από αρχείο που όμως δεν βρίσκεται στην αναμενόμενη θέση. Η αντιμετώπιση τέτοιων απροσδόκητων συμβάντων γίνεται με την αξιοποίηση του μηχανισμού των εξαιρέσεων (exceptions).

Ας βάλουμε όμως τα θέματα σε μια σειρά.

16.1 Η στοίβα κλήσης

Έχουμε ήδη δει πώς μπορούμε να χρησιμοποιούμε τη λέξη throw για να παράγουμε μια εξαίρεση τύπου RuntimeException. Ας το δούμε λίγο πιο συγκεκριμένα με τη βοήθεια του κώδικα 16.1

```
public class CallStackUn {

    static int a() {
        b();
        return 0;
    }

    static int b() {
        c();
        return 0;
    }

    static void c() {
        throw new RuntimeException("exception occured in function c");
    }

    public static void main(String[] args) {
//openfiles
        a();
    }
}
```

Κώδικας 16.1 *Παραγωγή RuntimeException*

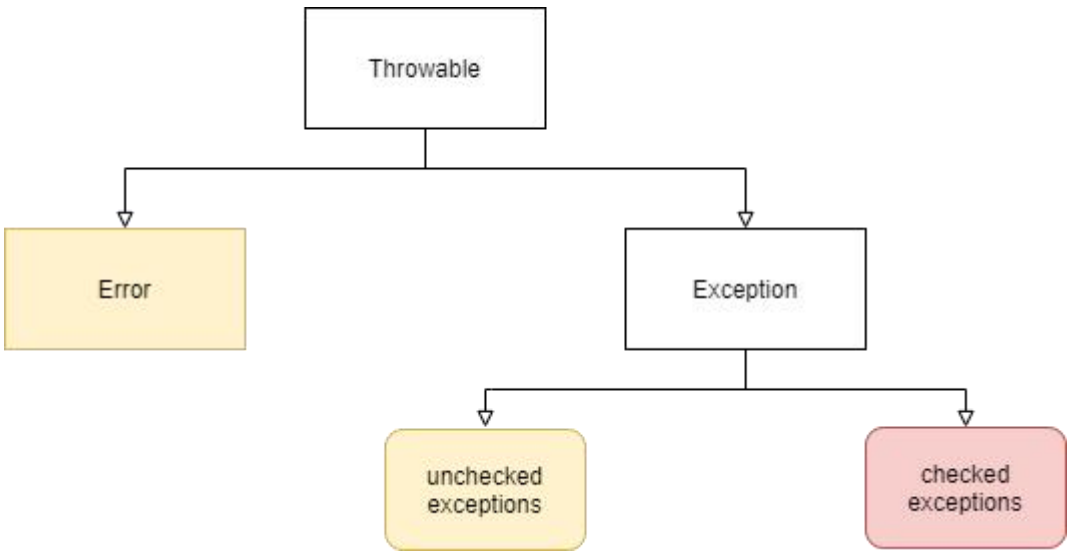
Στον κώδικα 16.1, η main καλεί την a, η a καλεί την b, η b την c και τέλος η c με χρήση της λέξης-κλειδί throw παράγει μια εξαίρεση τύπου RuntimeException [1]. Ενδιαφέρον έχει η έξοδος του κώδικα που έχει ως εξής:

```
Exception in thread "main" java.lang.RuntimeException:
exception occurred in function c
     at packagename.CallStackUn.c(CallStackUn.java:21)
     at packagename.CallStackUn.b(CallStackUn.java:16)
     at packagename.CallStackUn.a(CallStackUn.java:11)
     at packagename.CallStackUn.main(CallStackUn.java:25)
C:\Users\lmous\AppData\Local\NetBeans\Cache\12.4\executor-
snippets\run.xml:111: The following error occurred while
executing this line:
C:\Users\lmous\AppData\Local\NetBeans\Cache\12.4\executor-
snippets\run.xml:68: Java returned: 1
```

Η έξοδος αυτή μας πληροφορεί ότι εκδηλώθηκε μια εξαίρεση στην εφαρμογή μας και εμφανίζει τη στοίβα κλήσης (the call stack). Διαβάζοντας τη στοίβα κλήσης βλέπουμε πως η εξαίρεση προκλήθηκε στη γραμμή 21 μέσα στη συνάρτηση c που κλήθηκε από την β στη γραμμή 16 η οποία είχε κληθεί από την a στη γραμμή 11 που με τη σειρά της κλήθηκε από την main στην γραμμή 25 του κώδικα.

16.2 Τύποι απροσδόκητων λαθών

Η Java διαθέτει ένα μεγάλο σύνολο από προκαθορισμένους τύπους απροσδόκητων λαθών. Όπως θα δούμε στη συνέχεια αυτής της ενότητας υπάρχει η δυνατότητα, η εφαρμογή να ορίσει τους δικούς της τύπους. Όλες οι κλάσεις που αναπαριστούν τύπους λαθών, προκαθορισμένους ή οριζόμενους από τον χρήστη είναι παράγωγες της κλάσης Throwable.



Σχήμα 16.1 *Οργάνωση των τύπων απροσδόκητων λαθών*

Όπως φαίνεται στο σχήμα 16.1 οι τύποι των απροσδόκητων λαθών χωρίζονται καταρχάς σε δύο μεγάλες κατηγορίες, στα Errors και στα Exceptions. Τα δε Exceptions χωρίζονται περαιτέρω σε ελεγχόμενα (checked) και προαιρετικά ελεγχόμενα (unchecked). Ο μεταγλωττιστής υποχρεώνει τον προγραμματιστή να διαχειριστεί τις ελεγχόμενες εξαιρέσεις. Αντίθετα, η διαχείριση των προαιρετικών εξαιρέσεων δεν επιβάλλεται από τον μεταγλωττιστή. Παρόμοια, ο προγραμματιστής δεν υποχρεώνεται να διαχειριστεί λάθη τύπου Error ή κλάσεων παράγωγων της Error. Ας σημειωθεί πως στο σχήμα 16.1, το τμήμα Error αντιπροσωπεύει και όλη την ιεραρχία των παράγωγων κλάσεων της κλάσης Error. Παρόμοια, το τμήμα unchecked exceptions

αντιπροσωπεύει την ιεραρχία των προαιρετικά ελεγχόμενων εξαιρέσεων και το τμήμα checked exceptions την ιεραρχία των ελεγχόμενων εξαιρέσεων.

Παραδείγματα κλάσεων παράγωγων της Error είναι οι κλάσεις IOError και η OutOfMemoryError. Λάθος τύπου IOError μπορεί να παραχθεί όταν ένα σοβαρό και μη αντιμετωπίσιμο πρόβλημα εκδηλώνεται στο σύστημα Εισόδου/Εξόδου. Το OutofMemoryError παράγεται στην περίπτωση που η εικονική μηχανή της Java εξαντλήσει τη διαθέσιμη μνήμη. Και στις δύο περιπτώσεις, η εφαρμογή δεν μπορεί να κάνει οτιδήποτε άλλο παρά να τερματίσει άμεσα τη λειτουργία της. Γενικότερα, Errors παράγονται από λάθη που οφείλονται σε παράγοντες εξωτερικούς στην εφαρμογή για τα οποία η εφαρμογή δεν μπορεί να υιοθετήσει αποτελεσματικό τρόπο αντιμετώπισης.

Παραδείγματα προαιρετικά ελεγχόμενων εξαιρέσεων συνιστούν οι κλάσεις NullPointerException, ArrayIndexOutOfBoundsException και IllegalArgumentException. Οι μη ελεγχόμενες αφορούν λάθη που οφείλονται στην εφαρμογή αλλά είναι τέτοιος ο χαρακτήρας τους που η εφαρμογή δεν μπορεί να κάνει κάτι παρά να τερματίσει. Για παράδειγμα, αν η εφαρμογή μας παράγει λάθος τύπου NullPointerException, τότε κάποιο λάθος (bug) έχει ο κώδικάς μας, οπότε αναγκαστικά θα διακοπεί η εκτέλεση της εφαρμογής και θα προσπαθήσουμε να εντοπίσουμε και να διορθώσουμε το λάθος. Μη ελεγχόμενες εξαιρέσεις είναι όλες οι παράγωγες κλάσεις της RuntimeException.

Χαρακτηριστικά παραδείγματα ελεγχόμενων εξαιρέσεων αποτελούν οι κλάσεις IOExeption και FileNotFoundException. FileNotFoundException παράγεται όταν η εφαρμογή προσπαθεί να προσπελάσει ένα αρχείο και αυτό δεν βρεθεί στη δεδομένη διαδρομή, ενώ IOException μπορεί να παραχθεί όταν για παράδειγμα επιχειρήσουμε να γράψουμε σε ένα αρχείο στο οποίο επιτρέπεται μόνο η ανάγνωση. Και στις δύο περιπτώσεις είναι δυνατόν η εφαρμογή να μπορεί να αποκριθεί. Για παράδειγμα, μπορεί το FileNotFoundException να προκλήθηκε από λανθασμένη επιλογή διαδρομής από τον χρήστη. Σε αυτήν την περίπτωση, μπορεί η εφαρμογή να βγάλει ένα μήνυμα προς τον χρήστη και να ζητήσει τη σωστή διαδρομή. Γενικότερα, οι ελεγχόμενες εξαιρέσεις αφορούν λάθη από τα οποία μια σωστά σχεδιασμένη εφαρμογή έχει τη δυνατότητα να ανακάμψει. Ελεγχόμενες εξαιρέσεις αναπαριστούν όλες οι κλάσεις που είναι παράγωγες της κλάσης Exception και δεν είναι παράγωγες της RuntimeError.

16.3 Σύλληψη εξαιρέσεων

Όπως αναφέραμε τις ελεγχόμενες εξαιρέσεις μας υποχρεώνει ο μεταγλωττιστής να τις διαχειριστούμε. Για μια ελεγχόμενη εξαίρεση που παράγεται μέσα σε μία συνάρτηση, έχουμε δύο δυνατότητες: Να τη συλλάβουμε για να τη διαχειριστούμε περαιτέρω ή απλώς να την προωθήσουμε. Στον κώδικα 16.2 προωθούμε την εξαίρεση τύπου Exception που παράγεται στη συνάρτηση c. Η προώθηση επιτυγχάνεται καταρχάς από την απουσία κώδικα που θα τη συλλάμβανε και από τη λέξη-κλειδί throws που ακολουθεί τη λίστα παραμέτρων της c συνοδευόμενη από το όνομα της εξαίρεσης που προωθείται.

```
public class CallStackCh {

    static int a() throws Exception {
        b();
        return 0;
    }

    static int b() throws Exception {
        c();
        return 0;
    }

    static void c() throws Exception {
        throw new Exception("exception occured in function c");
    }

    public static void main(String[] args) throws Exception {
        a();
    }
}
```

Κώδικας 16.2 *Αντιμετώπιση ελεγχόμενων εξαιρέσεων ως μη ελεγχόμενες.*

Το ερώτημα είναι τώρα τι σημαίνει προώθηση, δηλαδή προς τα πού προωθείται η εξαίρεση. Η εξαίρεση ακολουθεί όλη τη στοίβα κλήσης σε αντίστροφη κατεύθυνση είτε έως ότου κάποια συνάρτηση από τη στοίβα κλήσης συλλάβει την εξαίρεση ή έως τον τερματισμό της εφαρμογής και την εκτύπωση της στοίβας κλήσης. Έτσι στον κώδικα 16.2, η εξαίρεση μεταβιβάζεται στην b. Εκεί μπορεί να υπάρχει κώδικας που τη συλλαμβάνει ή απλώς μπορεί να προωθείται και πάλι. Στο παράδειγμά μας προωθείται. Πού; Στη συνάρτηση από την οποία κλήθηκε η b, δηλαδή στην a. Η δε a προωθεί και πάλι στην main και η main τερματίζει την εφαρμογή με τιμή επιστροφής 1 και προκαλεί εκτύπωση της στοίβας κλήσης. Αν τρέξετε τον κώδικα 16.2, θα διαπιστώσετε πως η έξοδός του είναι παρόμοια με αυτήν του κώδικα 16.1. Αυτό συμβαίνει, γιατί παρότι στον κώδικα 16.2 έχουμε ελεγχόμενη εξαίρεση δεν τη συλλαμβάνουμε αλλά την επαναπροωθούμε σε όλες τις συναρτήσεις της στοίβας κλήσης.

Ας δούμε τώρα πώς συλλαμβάνουμε μια εξαίρεση. Σε οποιαδήποτε συνάρτηση της στοίβας κλήσης μπορεί η εξαίρεση να συλληφθεί. Η σύλληψη της εξαίρεσης γίνεται με τη βοήθεια ενός μπλοκ try…catch. Αντικαταστήστε στον κώδικα 16.2 τη συνάρτηση a με τη συνάρτηση a που δίνεται στον κώδικα 16.3.

```
static int a() {
    try {
        b();
    } catch (Exception e) {
        System.out.println(
                e.getClass() + " " + e.getMessage() + " caught in a");
    }
    return 0;
}
```

Κώδικας 16.3 *Σύλληψη της εξαίρεσης στη συνάρτηση a.*

Η συνάρτηση a δεν προωθεί πλέον την εξαίρεση αλλά τη συλλαμβάνει. Έτσι η λέξη-κλειδί throws λείπει από την επικεφαλίδα της συνάρτησης, ενώ ο κώδικας από όπου μπορεί να παραχθεί εξαίρεση μέσα στην a, δηλαδή η συνάρτηση b περικλείεται μέσα σε try…catch μπλοκ. Μέσα στο try μπλοκ τοποθετείται ο κώδικας που μπορεί να παράγει εξαίρεση. Στη συνέχεια ακολουθεί το catch μπλοκ που μπορεί να συλλάβει την εξαίρεση εφόσον ελέγχει τον τύπο της. Όπως θα δούμε στη συνέχεια, ένα try μπλοκ μπορεί να ακολουθείται από πολλαπλά catch μπλοκ καθένα από τα οποία συλλαμβάνει διαφορετικές εξαιρέσεις. Τα catch μπλοκ είναι γνωστά ως διαχειριστές εξαιρέσεων (exception handlers). Στον κώδικα 16.3 ελέγχεται μία εξαίρεση τύπου Exception δηλαδή του τύπου που παρήχθη από την c, επομένως συλλαμβάνεται.

Τι είναι όμως αυτό που παράγεται στην c και στη συνέχεια προωθείται από συνάρτηση σε συνάρτηση ώσπου να συλληφθεί στην a; Είναι ένα αντικείμενο τύπου Exception. Στο catch μπλοκ που γίνεται η σύλληψη της εξαίρεσης το αντικείμενο αυτό είναι διαθέσιμο. Το μόνο που χρειάζεται να κάνει ο προγραμματιστής είναι να το ονομάσει ώστε να μπορεί να το χρησιμοποιήσει. Στον κώδικα 16.3, το αντικείμενο έχει ονομαστεί e. Στη συνέχεια, μέσα στην catch, χρησιμοποιώντας το αναγνωριστικό e αναφερόμαστε στο αντικείμενο τύπου Exception που παρήχθη στην c. Το αντικείμενο e μεταφέρει πληροφορίες που αφορούν το λάθος που συνέβη στην c.

Το μήνυμα που θα τυπωθεί τώρα είναι

```
class java.lang.Exception exception occured in function c
caught in a
```

Όπως είπαμε οποιαδήποτε συνάρτηση στη στοίβα κλήσης δύναται να συλλάβει την εξαίρεση εφόσον διαθέτει το κατάλληλο try…catch μπλοκ. Ας αντικαταστήσουμε τώρα και τη συνάρτηση b με τον κώδικα 16.4.

```
static int b() throws Exception {
    try {
        c();
    } catch (Exception e) {
        System.out.println(
                e.getClass() + " " + e.getMessage() + " caught in b");
    }
```

```
        c();
        return 0;
    }
```

Κώδικας 16.4 *Σύλληψη της εξαίρεσης στη συνάρτηση b.*

Το μήνυμα που λαμβάνουμε τώρα είναι

```
class java.lang.Exception exception occured in function c
caught in b
```

Αυτό είναι αναμενόμενο, καθώς η εξαίρεση προωθείται πρώτα στην b και μετά στην a. Από τη στιγμή που η b συλλαμβάνει την εξαίρεση, η προώθησή της καταστέλλεται. Παρομοίως, θα μπορούσε να συλληφθεί η εξαίρεση στη συνάρτηση c όπως δείχνει ο κώδικας 16.5:

```
    static void c() throws Exception {
        try {
            throw new Exception("exception occured in function c");
        } catch (Exception e) {
            System.out.println(
                    e.getClass() + " " + e.getMessage() + " caught in c");
        }
    }
```

Κώδικας 16.5 *Σύλληψη της εξαίρεσης στη συνάρτηση c όπου παρήχθη αρχικά.*

Από όσα παραθέσαμε μέχρι τώρα φαίνεται πως ο μηχανισμός των εξαιρέσεων μας δίνει τη δυνατότητα να συνδυάσουμε πληροφορίες που είναι γνωστές στον κώδικα που παράγει την εξαίρεση με πληροφορίες στον κώδικα που τη συλλαμβάνει. Με άλλα λόγια, παρέχεται η δυνατότητα να συνδυαστούν πληροφορίες που αφορούν το λάθος που προκάλεσε την εξαίρεση με πληροφορίες διαθέσιμες στο περιβάλλον κλήσης του κώδικα που συμβαίνει το λάθος. Αυτή η δυνατότητα είναι συχνά αναγκαία για την ορθή αντιμετώπιση ενός απροσδόκητου λάθους.

16.4 Η πρόταση finally

Ας δούμε τώρα μια εφαρμογή διαχείρισης αρχείων στην οποία αξιοποιείται η διαχείριση εξαιρέσεων:

```
    static void copyStudents(String source, String target) throws IOException {
        File input = new File(source);
        BufferedWriter out = null;
        Scanner scanner = null;
        int[] k = new int[2];

        try {
            scanner = new Scanner(input);
            FileWriter fW = new FileWriter(target);
            out = new BufferedWriter(fW);
            while (scanner.hasNextLine()) {
                String st = scanner.nextLine();
                out.write(st + "\n");
                k[2] = 1;
            }
        } catch (FileNotFoundException ex) {
            System.out.println(ex + " file " + source + " has not copied");
        } catch (IOException ex) {
            System.out.println(ex + " file " + source + " has not copied");
        }

        if (scanner != null) {
            scanner.close();
```

```
        }
        if (out != null) {
            out.close();
        }
    }
```

Κώδικας 16.6 *Αντιγραφή αρχείων με σύλληψη εξαιρέσεων*

Ο κώδικας 16.6 αντιγράφει το αρχείο που βρίσκεται στη θέση source στο αρχείο που προσδιορίζεται από τη θέση target. Καταρχάς, οι διαχειριστές των δύο αρχείων, δηλαδή οι μεταβλητές scanner και out αρχικοποιούνται στο null. Η αρχικοποίηση αυτή είναι απαραίτητη. Μέσα στην try επιχειρείται η σύνδεση των διαχειριστών των αρχείων με τα φυσικά αρχεία στον δίσκο με αξιοποίηση των δημιουργών της Scanner, της FileWriter και της BufferedWriter. Και οι τρεις δημιουργοί ενδέχεται να αποτύχουν είτε παράγοντας IOException είτε παράγοντας FileNotFoundException. Αν υποθέσουμε ότι μέσα στην try παράγεται IOException ή FileNotFoundException, ο κώδικας θα μεταβεί στην αντίστοιχη catch η οποία στην ουσία παράγει ένα μήνυμα και στη συνέχεια θα συνεχίσει η ροή προς το τελευταίο τμήμα της συνάρτησης όπου κλείνουμε τα αρχεία. Αν το λάθος συμβεί στον δημιουργό της Scanner, η μεταβλητή scanner ενδέχεται να μην έχει λάβει τιμή, η δε μεταβλητή out σίγουρα δεν θα έχει λάβει τιμή. Για να αποφύγουμε να καλέσουμε την close μέσω μιας αναρχικοποίητης μεταβλητής, αρχικοποιούμε τις scanner και out σε null, οπότε πριν τις χρησιμοποιήσουμε για κλείσιμο των αρχείων ελέγχουμε αν είναι null. Αν η τιμή κάποιου διαχειριστή είναι όντως null, σημαίνει πως το αντίστοιχο αρχείο δεν άνοιξε και επομένως δεν είναι αναγκαίο να το κλείσουμε. Εξάλλου η κλήση οποιασδήποτε μη στατικής συνάρτησης μέσω του null δεν είναι παρά ένα επιπλέον λάθος στον κώδικά μας. Αν είχαμε αφήσει τις μεταβλητές αναρχικοποίητες, ο έλεγχος αυτός θα ήταν αδύνατος. Προσέξτε πως δεν κλείνουμε τον FileWriter, καθώς αυτός αποτελεί υποκείμενη δομή του BufferedWriter το κλείσιμο του οποίου προκαλεί αυτόματα κλείσιμο και του FileWriter.

Μία άλλη σημαντική παρατήρηση είναι η εξής: Οι δύο εξαιρέσεις που αναφέραμε προηγουμένως συνδέονται μεταξύ τους με σχέσεις κληρονομικότητας. Πιο συγκεκριμένα, η FileNotFoundException είναι παράγωγη της IOException. Με άλλα λόγια, μια FileNotFoundException είναι μία IOException. Επομένως αν στο catch μπλοκ τοποθετήσουμε πρώτα την IOException και μετά την FileNotFoundException, τότε στην περίπτωση που θα παραχθεί FileNotFoundException αυτή θα συλληφθεί από τον διαχειριστή της IOException. Ευτυχώς, ο μεταγλωττιστής της Java εξετάζει τη σειρά που συλλαμβάνουμε τις συγγενικές εξαιρέσεις και δεν επιτρέπει τέτοιου είδους λάθη.

Προσέξτε πως στην επικεφαλίδα της copyStudent έχει προστεθεί το τμήμα throws IOException. Αυτό οφείλεται στο ότι IOException μπορεί να παραχθεί και από τις κλήσεις close στο τέλος του κώδικα. Στην περίπτωση που έχουμε εξαίρεση εκεί δεν μας είναι ιδιαίτερα χρήσιμη η διαχείρισή της, γιατί πολύ απλά δεν μπορούμε να κάνουμε κάτι για ένα τέτοιο σφάλμα. Από τη στιγμή που δεν διαχειριζόμαστε μια πιθανή εξαίρεση είμαστε υποχρεωμένοι να την προωθήσουμε στη στοίβα κλήσης.

Επίσης, η scanner.hasNextLine μπορεί να παράγει εξαίρεση τύπου IllegalStateException αν το αρχείο είναι κλειστό, η δε scanner.nextLine μπορεί να παράγει επίσης IllegalStateException ή NoSuchElementException στην περίπτωση που δεν υπάρχει άλλη γραμμή στο αρχείο. Οι εξαιρέσεις αυτές είναι και οι δύο προαιρετικά ελεγχόμενες. Θα μπορούσαμε βεβαίως να προσθέσουμε κώδικα ώστε να τις συλλάβουμε. Ωστόσο, στον συγκεκριμένο κώδικα αποκλείονται αυτές οι εξαιρέσεις, καθώς το αρχείο έχει ήδη ανοίξει αμέσως πριν κληθεί η hasNextLine και η nextLine έχει κληθεί αφού έχει προηγηθεί έλεγχος με την hasNextLine.

Παρόλα όσα έχουμε κάνει, αν τρέξουμε τον κώδικα 16.6 θα διαπιστώσουμε πως αυτός "πέφτει" εγείροντας μία εξαίρεση ArrayIndexOutOfBoundsException. Αυτό οφείλεται στην πρόταση k[2]=1. Ο πίνακας k έχει μήκος 2 και επομένως μόνο οι θέσεις του 0 και 1 είναι διαθέσιμες. Η απόπειρα να διαβάσουμε στη θέση 2 παράγει την εξαίρεση ArrayIndexOutOfBoundsException. Αυτή είναι προαιρετικά ελεγχόμενη. Θα μπορούσαμε να προσθέσουμε ένα catch για αυτήν την εξαίρεση. Ωστόσο πρόκειται για εξαίρεση που οφείλεται σε προγραμματιστικό λάθος. Μία τέτοια εξαίρεση μπορεί να παραχθεί από οπουδήποτε και δεν είναι καλή πρακτική να βάζουμε όλο τον κώδικα σε try μπλοκ για να συλλάβουμε μία τέτοια εξαίρεση ή άλλες μη ελεγχόμενες εξαιρέσεις. Επιπλέον, δεν μπορούμε να προσφέρουμε κάποια βοήθεια στον χρήστη κατά τον χρόνο εκτέλεσης. Το μόνο που μπορούμε να κάνουμε είναι να τερματίσουμε την εφαρμογή και να διορθώσουμε τον κώδικα, πράγμα για το οποίο δεν χρειάζεται σύλληψη της εξαίρεσης. Εξάλλου μετά την απολαθοποίηση του κώδικα τέτοια εξαίρεση δεν θα παράγεται.

Εκείνο όμως που αποτελεί πρόβλημα είναι πως η ArrayIndexOutOfBoundsException δεν επιτρέπει την ομαλή ροή του κώδικα και άρα την εκτέλεση των close. Το αποτέλεσμα είναι ότι τα αρχεία παραμένουν ανοιχτά και είναι πιθανή η απώλεια δεδομένων. Χρειαζόμαστε λοιπόν έναν μηχανισμό ο οποίος θα εγγυάται το κλείσιμο των αρχείων είτε παράγεται εξαίρεση οποιουδήποτε τύπου είτε όχι. Αυτόν ακριβώς τον μηχανισμό προσφέρει η Java με τη δεσμευμένη λέξη finally [2]. Όπως δείχνει ο κώδικας 16.7, ένα μπλοκ finally ακολουθεί τα μπλοκ catch. Μέσα στο μπλοκ finally τοποθετούμε κώδικα που θέλουμε να εκτελεστεί έτσι κι αλλιώς, είτε παραχθεί εξαίρεση είτε όχι.

```
    static void copyStudentsF(String source, String target) throws IOException {
        File input = new File(source);
        BufferedWriter out = null;
        Scanner scanner = null;
        int[] k = new int[2];

        try {
            scanner = new Scanner(input);
            FileWriter fW = new FileWriter(target);
            out = new BufferedWriter(fW);
            while (scanner.hasNextLine()) {
                String st = scanner.nextLine();
                out.write(st + "\n");
                k[2] = 1;
            }
        } catch (FileNotFoundException ex) {
            System.out.println(ex + " file " + source + " has not copied");
        } catch (IOException ex) {
            System.out.println(ex + " file " + source + " has not copied");
        } finally {
            if (scanner != null) {
                scanner.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
```

Κώδικας 16.7 *Σύλληψη εξαιρέσεων με μπλοκ finally*

Ένα θέμα που προκύπτει από τη χρήση του finally μπλοκ είναι πως στην περίπτωση που παραχθεί εξαίρεση τόσο μέσα στο try μπλοκ όσο και μέσα στο finally μπλοκ, τότε η συνάρτηση στην οποία περικλείεται το try…catch…finally θα προωθήσει την εξαίρεση που παράχθηκε στο finally και όχι αυτή που παράχθηκε στο try. Αυτή που παράγεται στο try καταστέλλεται. Το θέμα αυτό μπορεί να αντιμετωπιστεί με τη δομή try-with-resources που προστέθηκε στην έκδοση 7.

```
    static void copyStudentsWR(String source, String target) throws IOException {
        File input = new File(source);

        try ( BufferedWriter out = new BufferedWriter(new FileWriter(target));
Scanner scanner = new Scanner(input);) {
            while (scanner.hasNextLine()) {
                String st = scanner.nextLine();
                out.write(st + "\n");
            }
        } catch (FileNotFoundException ex) {
            System.out.println(ex + " file " + source + " has not copied");
        } catch (IOException ex) {
            System.out.println(ex + " file " + source + " has not copied");
        }
    }
```

Κώδικας 16.8 *try with resources*

Στον κώδικα 16.8 χρησιμοποιούμε την try-with-resources [3]. Εκείνο που περιλαμβάνει επιπλέον της απλής try είναι οι παρενθέσεις μεταξύ της δεσμευμένης λέξης try και του μπλοκ που ακολουθεί την try. Μέσα σε αυτές τις παρενθέσεις δημιουργούμε τα απαιτούμενα αντικείμενα μέσω των οποίων θα διαχειριστούμε πόρους, όπως τα αρχεία στο παράδειγμά μας. Οποιοδήποτε αντικείμενο ανήκει σε κλάση που υλοποιεί τη διεπαφή AutoCloseable μπορεί να δημιουργηθεί μέσα στις παρενθέσεις. Στη συνέχεια, όλοι οι δεσμευμένοι πόροι θα κλείσουν αυτόματα. Με την try-with-resources σε περίπτωση που εκδηλωθεί εξαίρεση κατά το κλείσιμο των πόρων αλλά και μέσα στην try, η κύρια συνάρτηση θα προωθήσει την εξαίρεση που εκδηλώθηκε στην try.

16.5 Εξαιρέσεις οριζόμενες από τον χρήστη

Αν μεταβείτε στην τεκμηρίωση της κλάσης Exception, θα διαπιστώσετε πως αυτή διαθέτει δεκάδες παράγωγες κλάσεις. Μάλιστα, κάθε μία διαθέτει αρκετά περιγραφικό και μακρύ όνομα. Γενικά, στις βιβλιοθήκες της Java έχει επικρατήσει η τάση για κάθε διαφορετικό τύπο λάθους να ορίζεται μία αντίστοιχη κλάση εξαιρέσεων. Έτσι στις περισσότερες περιπτώσεις ο τύπος και μόνο της παραγόμενης εξαίρεσης μας δίνει αρκετές πληροφορίες για το λάθος που την προκάλεσε.

Είτε θελήσουμε να ακολουθήσουμε στις εφαρμογές μας αυτήν τη σχεδιαστική προσέγγιση είτε όχι, συχνά θα προκύψει η ανάγκη να ορίσουμε νέους τύπους εξαιρέσεων. Για παράδειγμα, έστω ότι διαβάζουμε τα στοιχεία των σπουδαστών από οριοθετημένο αρχείο και όλες οι λειτουργίες επιτελούνται σωστά. Καμία εξαίρεση τύπου IOException ή FileNotFoundException δεν παράγεται και το αρχείο διαβάζεται γραμμή προς γραμμή με επιτυχία. Σε κάθε γραμμή όμως του αρχείου περιμένουμε ανάμεσα σε άλλα και 5 βαθμούς που διαχωρίζονται μεταξύ τους με ένα διάστημα, π.χ. 10.0 11.0 12.0 13.0 14.0 15.0. Αντί όμως της αναμενόμενης τιμής σε μια γραμμή του αρχείου, έχουμε 2 βαθμούς που δεν διαχωρίζονται μεταξύ τους, π.χ. 10.0 11.0 12.0 13.014.0 15.0. Σε αυτήν την περίπτωση, μπορούμε να χρησιμοποιήσουμε μια προκαθορισμένη γενική εξαίρεση, όπως η Exception ή να ορίσουμε δικό μας τύπο εξαίρεσης.

Για να ορίσουμε μια προαιρετικά ελεγχόμενη θα πρέπει να δημιουργήσουμε μια κλάση που είναι παράγωγη της RuntimeException ή παράγωγη κάποιας κλάσης απογόνου της RuntimeException. Για να ορίσουμε ελεγχόμενη εξαίρεση θα πρέπει να δημιουργήσουμε μία κλάση παράγωγη της Exception ή κάποιου απογόνου της Exception εξαιρουμένης της RuntimeException και των απογόνων της.

```
public class MallFormedFileLine extends Exception {
    private final int line;

    public MallFormedFileLine(int line) {
        this.line = line;
    }

    public int getLine() {
        return line;
    }

    @Override
    public String toString() {
        return "MallFormedFileLine{" + "line=" + line + '}';
    }
}
```

Κώδικας 16.9 *Παράδειγμα εξαίρεσης οριζόμενης από τον χρήστη*

Στον κώδικα 16.9 η κλάση MallFormedFileLine είναι παράγωγη της κλάσης Exception. Η MallFormedFileLine σχεδιάστηκε έτσι ώστε να μπορεί να χρησιμοποιηθεί σε παραλλαγή της readStudents του κώδικα 15.3 όπου σε ένα οριοθετημένο αρχείο κάποια γραμμή δεδομένων ενδέχεται να μην είναι διαμορφωμένη όπως αναμένεται.

```
static ArrayList<Student> readStudents(String fName) {
    ArrayList<Student> rVal = new ArrayList<>();
    File input = new File(fName);
```

```
        int ln = 0;
        try ( Scanner scanner = new Scanner(input);) {
            while (scanner.hasNextLine()) {
                ln++;
                String st = scanner.nextLine();
                String[] stFields = st.split(" ");
                if (stFields.length != 10) {
                    throw new MallFormedFileLine(ln);
                }
                Student student = create(stFields);
                rVal.add(student);
            }
        } catch (FileNotFoundException ex) {
            System.out.println(ex + ": " + fName);
        } catch (MallFormedFileLine e) {
            System.out.println("Format Error at line: " + e.getLine());
        }
        return rVal;
    }
```

Κώδικας 16.10 *Αξιοποίηση της MallFormedFileLine*

Κάθε γραμμή του γνωστού μας αρχείου studentsd.dat αναμένεται να διαχωρίζεται σε 10 τμήματα. Όταν αυτό δεν συμβαίνει, τότε έχουμε κάποιο λάθος στη γραμμογράφηση του αρχείου. Σε αυτήν την περίπτωση μπορούμε να παράγουμε μια MallFormedFileLine όπως φαίνεται στον κώδικα 16.10. Η MallFormedFileLine μας πληροφορεί σε ποια ακριβώς γραμμή του αρχείου υπάρχει το πρόβλημα.

16.6 Διαχείριση και σύνθετη σύλληψη

Στον κώδικα 16.10 με τη σύλληψη της MallFormedFileLine διακόπηκε η φόρτωση των στοιχείων των σπουδαστών στο ArrayList. Ως αποτέλεσμα, το ArrayList τα στοιχεία του οποίου εμφανίζονται στην main περιέχει μόνο τα στοιχεία των σπουδαστών που προηγούνται της γραμμής που προκάλεσε την εξαίρεση. Ανάλογα όμως με τη φύση της εφαρμογής είναι δυνατόν η φόρτωση του αρχείου να χρειάζεται να συνεχιστεί με τις υπόλοιπες γραμμές του αρχείου εφόσον έχουν σωστή γραμμογράφηση.
Ένα άλλο θέμα που δεν αντιμετωπίζει ο κώδικας 16.10 είναι η εξαίρεση NumberFormatException. Πρόκειται για μη ελεγχόμενη εξαίρεση που ενδέχεται να παραχθεί από κάποια κλήση Double.parseDouble μέσα στην create. Τέτοια εξαίρεση θα παραχθεί όταν ένα String δεν μπορεί να μετατραπεί σε double. Για παράδειγμα, μια γραμμή του αρχείου έχει την ακόλουθη μορφή:

```
212 Jack Green 11.0 12.0 13.0 14.0 15.0 16.0 2005-05-01
```

Οι δύο εξαιρέσεις, η MallFormedFileLine και η NumberFormatException αναφέρονται σε παρόμοια περίπου λάθη και ενδεχομένως να θέλουμε να χρησιμοποιήσουμε τον ίδιο διαχειριστή και για τις δύο.
Στον κώδικα 16.11 η νέα έκδοση της readStudents δίνει απαντήσεις σε αυτά τα ζητήματα:

```
    static ArrayList<Student> readStudentsC(String fName) {
        ArrayList<Student> rVal = new ArrayList<>();
        File input = new File(fName);
        Scanner scanner = null;
        Scanner keyboard=new Scanner(System.in);
        try {
            scanner = new Scanner(input);
        } catch (FileNotFoundException ex) {
            System.out.println(ex + ": " + fName);
            System.exit(1);
        }
        int ln = 0;
        while (true) {
            ln++;
```

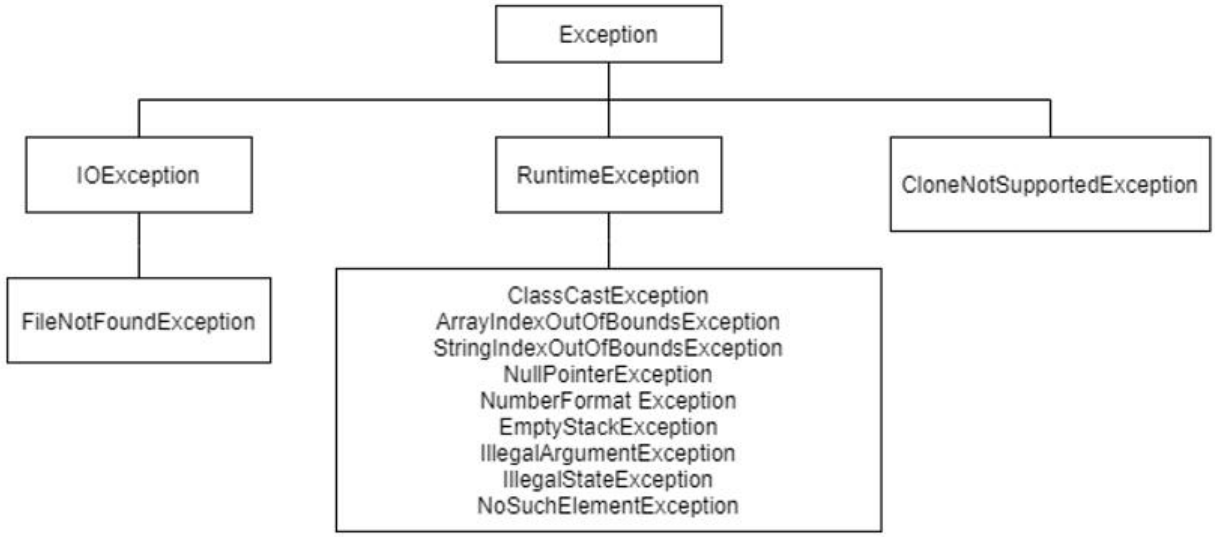
```
            try {
                if (!scanner.hasNextLine())
                    break;
                String st = scanner.nextLine();
                String[] stFields = st.split(" ");
                if (stFields.length != 10) {
                    throw new MallFormedFileLine(ln);
                }
                Student student = create(stFields);
                rVal.add(student);

            } catch (MallFormedFileLine | NumberFormatException e) {
                System.out.println(e);
                System.out.println("Continue reading Y/N? ");
                String userAnswer = keyboard.nextLine();
                if (userAnswer.equals("N")) {
                    scanner.close();
                    System.exit(1);
                }
            }
        }
        scanner.close();
        return rVal;
    }
```

Κώδικας 16.11 *Διαχείριση και σύνθετη σύλληψη*

Στον κώδικα 16.11 καταρχάς συνδέουμε ένα αντικείμενο Scanner με τη ροή τυπικής εξόδου. Η ροή αυτή είναι μονίμως ανοικτή, εφόσον δεν την κλείσει η εφαρμογή μας και έτσι η σύνδεση της Scanner δεν κινδυνεύει να αποτύχει, οπότε την πραγματοποιούμε έξω από την try. Στη συνέχεια μέσα στην try ανοίγουμε το αρχείο εισόδου, δηλαδή το αρχείο που είναι αποθηκευμένα τα στοιχεία των μαθητών. Αν το άνοιγμα του αρχείου αποτύχει, τότε απλώς βγάζουμε ένα μήνυμα και καλούμε την System.exit με παράμετρο 1. Η χρήση της System.exit θέλει προσοχή, καθώς προκαλεί άμεσο τερματισμό της εικονικής μηχανής και της εφαρμογής μας. Επομένως, μετά την κλήση της δεν είναι δυνατόν να γίνει οριστικοποίηση πόρων. Στο συγκεκριμένο παράδειγμα όμως εφόσον το αρχείο δεν βρέθηκε, δεν χρειάζεται και να κλείσει. Οπότε αν η εφαρμογή συνεχίσει, είμαστε βέβαιοι πως η σύνδεση μεταξύ αρχείου και scanner έχει επιτευχθεί. Στη συνέχεια, τοποθετούμε την try μέσα σε μια επαναληπτική διαδικασία η οποία διακόπτεται όταν τελειώσουν οι γραμμές του αρχείου. Αν παραχθεί εξαίρεση, έχουμε έναν διαχειριστή και για τους δύο τύπους που πιθανώς θα παραχθούν. Αυτό επιτυγχάνεται με την κάθετο που χωρίζει τους τύπους των εξαιρέσεων και είναι γνωστό ως piping. Στην περίπτωση που εφαρμόζουμε piping, ας έχουμε υπόψη πως το αντικείμενο e είναι final, ενώ αν δεν εφαρμόζεται piping δεν είναι. Στη συνέχεια εμφανίζουμε το αντικείμενο που παρέχει ικανοποιητική πληροφόρηση σε σχέση με το λάθος και ρωτάμε τον χρήστη αν επιθυμεί να συνεχίσει η φόρτωση του αρχείου ή όχι. Αν απαντήσει αρνητικά, κλείνουμε το αρχείο και καλούμε την System.exit(1), διαφορετικά εκτελείται το επόμενο βήμα επανάληψης που επιχειρεί να διαβάσει την επόμενη γραμμή του αρχείου.

16.7 Βασικές προκαθορισμένες εξαιρέσεις

Η ιεραρχία των εξαιρέσεων της Java περιλαμβάνει πολλές δεκάδες υποκλάσεις. Σε αυτήν την ενότητα παραθέτουμε ένα υποσύνολο που περιλαμβάνει εξαιρέσεις που χρησιμοποιούνται συχνά, πάντα στο πλαίσιο της μέχρι εδώ διδασκόμενης ύλης.



Σχήμα 16.2 *Ιεραρχία βασικών προκαθορισμένων εξαιρέσεων*

Η κλάση Exception στην κορυφή της ιεραρχίας είναι η γονική κλάση όλων των εξαιρέσεων στην Java. Συνήθως, όταν θέλουμε να ορίσουμε δικό μας τύπο εξαίρεσης, αυτήν την κλάση κληρονομούμε.

Η κλάση IOException είναι η γενική εξαίρεση που δείχνει πως κάποιο είδος προβλήματος έχει συμβεί κατά την είσοδο ή έξοδο δεδομένων στην εφαρμογή.

Εξαίρεση τύπου FileNotFoundException παράγεται από τους δημιουργούς των κλάσεων FileInputStream, FileOutputStream, and RandomAccessFile όταν το αρχείο με το οποίο προσπαθούν να συνδεθούν δεν βρίσκεται στην προκαθορισμένη διαδρομή ή κάποια άλλη αιτία δεν επιτρέπει το άνοιγμά του.

CloneNotSupportedException παράγεται όταν επιχειρήσουμε να επανορίσουμε τη συνάρτηση clone σε μία κλάση που δεν υλοποιεί τη διεπαφή Cloneable.

Οι υπόλοιπες εξαιρέσεις του σχήματος 16.2 είναι μη ελεγχόμενες.

Η RuntimeException είναι η γονική κλάση όλων των μη ελεγχόμενων εξαιρέσεων της Java.

ClassCastException παράγεται όταν επιχειρηθεί η μετατροπή του τύπου ενός αντικειμένου σε μη συμβατή κλάση.

ArrayIndexOutOfBoundsException παράγεται όταν η εφαρμογή επιχειρήσει να διαβάσει ή να γράψει έξω από τα όρια ενός πίνακα.

StringIndexOutOfBoundsException παράγεται όταν η εφαρμογή επιχειρήσει να διαβάσει χαρακτήρα ή άλλο τμήμα ενός String αλλά υπερβεί τα όρια του String.

NullPointerException παράγεται όταν διαμέσου μεταβλητής αναφοράς που έχει την τιμή null επιχειρηθεί η κλήση μεθόδου ή η προσπέλαση μεταβλητής στιγμιότυπου. Λάβετε όμως υπόψη ότι δια μέσου μεταβλητής αναφοράς με τιμή null είναι επιτρεπτή η κλήση στατικών μελών χωρίς πρόβλημα αρκεί βέβαια να πρόκειται για μέλη της κλάσης στην οποία αναφέρεται η μεταβλητή.

NumberFormatException παράγεται όταν επιχειρηθεί η μετατροπή ενός String σε αριθμό αλλά το String δεν είναι μετατρέψιμο, π.χ. “120!”.

EmptyStackException παράγεται όταν επιχειρηθεί pop από άδειο Stack.

IllegalArgumentException παράγεται όταν σε συνάρτηση περνάμε λανθασμένα πραγματικές παραμέτρους. Για παράδειγμα, στον κώδικα 7.14, η factorial παράγει RuntimeException όταν κληθεί με αρνητική παράμετρο. Θα μπορούσε η RuntimeException να αντικατασταθεί με IllegalArgumentException που είναι πιο εξειδικευμένη σε σχέση με το συγκεκριμένο λάθος.

IllegalStateException παράγεται όταν επιχειρούμε ενέργειες σε αντικείμενο Scanner το οποίο είναι κλειστό και σε άλλες αντίστοιχες περιπτώσεις.

NoSuchElementException παράγεται όταν επιχειρήσουμε να διαβάσουμε δεδομένα που δεν υπάρχουν. Για παράδειγμα, αν καλέσουμε την next ενός Iterator χωρίς να έχουμε ελέγξει με την hasNext ή αν καλέσουμε την nextLine ενός Scanner χωρίς να έχουμε ελέγξει με την hasNextLine είναι πιθανή η παραγωγή της.

16.8 Λυμένες Ασκήσεις

16.8.1 Οι εξαιρέσεις ως μέρος της κανονικής ροής

Να αναπτύξετε συνάρτηση που μετράει πόσες φορές υπάρχει κάθε γράμμα του αγγλικού αλφάβητου, ανεξαρτήτως αν είναι κεφαλαίο ή πεζό, σε ένα αρχείο κειμένου. Να κληθεί η συνάρτηση στην main και να εμφανιστεί το αποτέλεσμα.

Λύση

Ο μηχανισμός των εξαιρέσεων μπορεί να χρησιμοποιηθεί και σε άλλες περιπτώσεις πέραν του ελέγχου των λαθών. Η λύση που προτείνουμε σε αυτήν την εφαρμογή αξιοποιεί τον μηχανισμό των εξαιρέσεων για να εξαιρέσει τους χαρακτήρες του κειμένου που δεν είναι γράμματα του αγγλικού αλφάβητου.

```
import java.io.File;
import java.util.Scanner;

public class CountLetters {

    public static String loadFile(String pathName) {
        StringBuilder sB = new StringBuilder();
        try ( Scanner input = new Scanner(new File(pathName))) {
            while (input.hasNextLine()) {
                sB.append(input.nextLine() + "\n");
            }
        } catch (Exception e) {
            System.out.println("File " + pathName + " not found");
            System.exit(1);
        }
        return new String(sB);
    }

    public static int[] countLetters(String data) {
        int[] counter = new int[26];
        data = data.toUpperCase();

        for (int i = 0; i < data.length(); i++) {
            try {
                counter[data.charAt(i) - 'A']++;
            } catch (Exception e) {

            }
        }
        return counter;
    }

    public static void main(String[] args) {
        String data = loadFile("data" + File.separator + "text01.txt");
        int[] counter = countLetters(data);
        for (int i = 0; i < counter.length; i++) {
            System.out.println((char) ('A' + i) + " " + counter[i]);
        }
    }
}
```

Κώδικας 16.12 *Εξαίρεση των χαρακτήρων που δεν είναι γράμματα του αγγλικού αλφάβητου*

Καταρχάς, η loadFile φορτώνει σε ένα String, τα δεδομένα του αρχείου που βρίσκεται στο pathName. Σε περίπτωση που αποτύχει, βγάζει κατάλληλο μήνυμα και τερματίζει η εφαρμογή.

Στη συνέχεια, η countLetters χρησιμοποιεί έναν πίνακα ακέραιων, μήκους 26, δηλαδή διαθέτει μία θέση για κάθε γράμμα του αγγλικού αλφάβητου. Στη συνέχεια, μετατρέπει όλους τους χαρακτήρες του

κειμένου σε κεφαλαίους. Χαρακτήρες που δεν διακρίνονται σε κεφαλαία ή πεζά, π.χ. η τελεία, δεν επηρεάζονται από αυτήν τη μετατροπή. Μετά, μέσα στην επαναληπτική διαδικασία προσπελαύνει ένα προς ένα τους χαρακτήρες και επιχειρεί να αυξήσει τον μετρητή στην αντίστοιχη θέση του αρχείου. Αν ωστόσο ο χαρακτήρας δεν αντιστοιχεί σε γράμμα του Αγγλικού αλφάβητου, τότε η έκφραση data.charAt(i) - 'A' βγαίνει έξω από τα όρια του πίνακα counter. Στην περίπτωση αυτή παράγεται ArrayOutOfIndexException. Ο διαχειριστής εξαιρέσεων που ακολουθεί συλλαμβάνει οποιαδήποτε εξαίρεση παραχθεί μέσα στο try. Όμως δεν κάνει κάποια περαιτέρω διαχείριση, απλώς με τη σύλληψή της η εξαίρεση καταστέλλεται. Έτσι η επαναληπτική διαδικασία συνεχίζεται και μετράει μόνο τους χαρακτήρες που μας ενδιαφέρουν.

Τέλος, στην main εκτυπώνονται οι χαρακτήρες μαζί με τον αντίστοιχο μετρητή.

16.8.2 Εξίσωση β' βαθμού

Να υλοποιηθεί η κλάση QuadraticEquation που αναπαριστά εξισώσεις δευτέρου βαθμού. Να υποστηρίζει μεθόδους υπολογισμού της διακρίνουσας και των ριζών x1 και x2. Να ελεγχθεί με κατάλληλες εξαιρέσεις. Στην main να αναπτυχθεί στοιχειώδης κώδικας ελέγχου.

Λύση

```
public class QuadraticEquation {

    private final double a, b, c;

    public QuadraticEquation(double a, double b, double c) {
        if (Lib.approximateEquals(a, 0)) {
            throw new IllegalArgumentException();
        }
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public QuadraticEquation(double a, double b) {
        this(a, b, 0);
    }

    public QuadraticEquation(double a) {
        this(a, 0, 0);
    }

    public double discriminant() {
        return Math.pow(b, 2) - 4 * a * c;
    }

    @Override
    public String toString() {
        return "QuadraticEquation{" + "a=" + a + ", b=" + b + ", c=" + c +
'}';
    }


    public double x1() {
        double disc = discriminant();
        if (disc < 0) {
            throw new RuntimeException(this+", discriminant < 0");
        }
        return (-b + Math.sqrt(disc)) / (2 * a);
    }

    public double x2() {
```

```
            double disc = discriminant();
            if (disc < 0) {
                throw new RuntimeException(this+", discriminant < 0");
            }
            return (-b - Math.sqrt(disc)) / (2 * a);
        }

        public static void main(String[] args) {
//          QuadraticEquation qE1=new QuadraticEquation(2,4,6),
//                  qE2=new QuadraticEquation(1,4),
//                  qE3=new QuadraticEquation(2);
//          System.out.println(qE1.discriminant());
//          System.out.println(qE2.discriminant());
//          System.out.println(qE3.discriminant());
//          System.out.println(qE1.x1()+" "+qE1.x2());
//          System.out.println(qE2.x1()+" "+qE2.x2());
//          System.out.println(qE3.x1()+" "+qE3.x2());

            QuadraticEquation qE=new QuadraticEquation(1, 1, -1);
            System.out.println(qE.discriminant());
            System.out.println(qE.x1()+" "+qE.x2());
        }
    }
```

Κώδικας 16.13 *Η κλάση QuadraticEquation*

Καταρχάς, έχουμε τρεις δημιουργούς στην κλάση ώστε εύκολα ο χρήστης της να μπορεί να δημιουργήσει εξισώσεις που όλοι οι συντελεστές είναι διάφοροι του μηδενός, εξισώσεις όπου ο συντελεστής c είναι 0 και εξισώσεις όπου b και c είναι μηδέν. Εξισώσεις στις οποίες ο b είναι μηδέν και ο c διάφορος του μηδενός δημιουργούνται αναγκαστικά με τον πρώτο δημιουργό, καθώς οι παράμετροι στην Java είναι παράμετροι θέσης (positional parameter) και δεν μπορεί να υποστηριχτεί διαφορετικός δημιουργός.

Στην εξίσωση δευτέρου βαθμού, ισχύει εξ ορισμού $a \neq 0$. Έτσι, αν επιχειρηθεί η δημιουργία αντικειμένου QuadraticEquation με $a==0$, ο δημιουργός παράγει μη ελεγχόμενη εξαίρεση τύπου IllegalArgumentException. Κάτι τέτοιο είναι πιθανό να συμβεί από κάποιο λάθος στον υπολογισμό του συντελεστή a σε άλλο σημείο της εφαρμογής. Ωστόσο, αν το λάθος φτάσει μέχρι τον δημιουργό της QuadraticEquation, δεν μπορούμε να κάνουμε πλέον κάτι και ως εκ τούτου επιλέξαμε την καταλληλότερη προαιρετικά ελεγχόμενη εξαίρεση την οποία δεν ελέγχουμε και προκαλεί τερματισμό της main.

Στον υπολογισμό των ριζών επίσης έχουμε τη μη ελεγχόμενη εξαίρεση RuntimeException. Σε μια τέτοια περίπτωση, θα πρέπει η τεκμηρίωση της κλάσης να αναφέρει ρητά πως οι x1 και x2 καλούνται μόνο μετά από έλεγχο πως η discriminant επιστρέφει αριθμό μεγαλύτερο ή ίσο με το 0. Παρόλα αυτά, ο χρήστης της κλάσης ενδέχεται να καλέσει την x1 ή την x2 χωρίς να ελέγξει. Για αυτήν την περίπτωση πρέπει να υπάρχει έλεγχος μέσα στις x1 και x2. Εδώ επίσης διακόπτουμε τη ροή της εφαρμογής. Μια εναλλακτική λύση θα ήταν να επιστρέφουμε Double.NaN.

Ο σχολιασμένος κώδικας μέσα στην main επιδεικνύει την παραγωγή εξαίρεσης (εφόσον αποσχολιαστεί) στην περίπτωση που ζητηθούν οι ρίζες, ενώ η διακρίνουσα είναι μικρότερη του μηδενός.

16.9 Ασκήσεις προς λύση

16.9.1 Άμεση πρόσβαση με διαχείριση εξαιρέσεων

Στον κώδικα 15.6, ανάγνωσης και εγγραφής σε αρχεία με τυχαία προσπέλαση, προσθέστε κατάλληλη διαχείριση των πιθανών εξαιρέσεων. Αντικαταστήστε τις εξαιρέσεις που ήδη περιλαμβάνονται στον κώδικα με καταλληλότερες.

16.9.2 Περιοδικός Πίνακας με διαχείριση εξαιρέσεων

Στην κλάση TableOfChemichalElements που υλοποιεί τη διεπαφή Cloneable προσθέστε διαχείριση εξαιρέσεων.

16.9.3 grep με διαχείριση εξαιρέσεων

Στη συνάρτηση public static void grep(File file, String searchItem) της άσκησης 15.8.2 προσθέστε κατάλληλη διαχείριση εξαιρέσεων.

16.9.4 Quiz με εξαιρέσεις οριζόμενες από τον χρήστη

Τροποποιήστε το quiz της άσκησης 15.8.1 ώστε να υποστηρίζει εξαιρέσεις. Ελέγξτε τα πιθανά λάθη στη δομή των ερωτήσεων με κατάλληλες εξαιρέσεις οριζόμενες από τον χρήστη.

Βιβλιογραφία

[1] C. Horstmann, Η γλώσσα προγραμματισμού Java, Αναλυτική Προσέγγιση. Broken Hill Publishers, 2021.

[2] J. Farrell, Java, Εκμάθηση με πρακτικά παραδείγματα. Κριτική, 2018.

[3] "The try-with-resources Statement (The Java™ Tutorials > Essential Java Classes > Exceptions)." https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html (accessed Jan. 12, 2022).

Κεφάλαιο 17

Σύνοψη
Στην ενότητα αυτή παρουσιάζονται οι γενικεύσεις της Java. Πιο συγκεκριμένα παρουσιάζονται οι γενικευμένες κλάσεις, οι γενικευμένες συναρτήσεις, τα όρια των γενικεύσεων, η κληρονομικότητα μεταξύ γενικευμένων τύπων και η αξιοποίηση του χαρακτήρα μπαλαντέρ.

Προαπαιτούμενη γνώση
Η ενσωμάτωση, η κληρονομικότητα και ο πολυμορφισμός στο αντικειμενοστρεφές μοντέλο με Java, οι αφηρημένες κλάσεις καθώς και οι βασικές προκαθορισμένες κλάσεις, δηλαδή οι String, StringBuilder, ArrayList, LinkedList, Stack, PriorityQueue, HashSet, HashMap, HashTree, LocalDate, Localtime, Period και Duration. Οι διεπαφές στην Java. Η βασική διαχείριση των αρχείων. Ο μηχανισμός των εξαιρέσεων.

Λέξεις κλειδιά
Γενίκευση (generic), παραμετροποιημένος τύπος (parametarized type), μπαλαντέρ (wildcard), πρωταρχικός τύπος (raw type).

17 Γενικεύσεις

Ήδη έχουμε χρησιμοποιήσει εκτεταμένα τις γενικεύσεις στην απλή μορφή τους. Οι γενικεύσεις αποτελούν ένα ισχυρό προγραμματιστικό εργαλείο που δίνει τη δυνατότητα ανάπτυξης γενικευμένων αλγορίθμων, δηλαδή αλγορίθμων που επενεργούν σε δεδομένα διαφορετικών τύπων, ενώ παράλληλα διατηρούν τον αυστηρό χαρακτήρα στον έλεγχο των τύπων. Επιπλέον διευκολύνουν την ανάπτυξη καθώς αφαιρούν την ανάγκη για μετατροπή τύπου και καθιστούν τον κώδικα ευανάγνωστο.

```
import java.util.ArrayList;

public class Benefits {

    public static void rawType() {
        ArrayList l = new ArrayList();
        l.add(1);
        l.add("Jim");
        l.add(new Object());
        int i = (Integer) l.get(0);
        String s = (String) l.get(1);
        Object o = l.get(2);
        System.out.println(i);
        System.out.println(s);
        System.out.println(o);
    }

    public static void generic() {
        ArrayList<String> l = new ArrayList<>();
        l.add("Jim");
        String s = l.get(0);
        System.out.println(s);
    }

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

Κώδικας 17.1 *Πρωταρχικός και γενικευμένος τύπος της ArrayList*

Στον κώδικα 17.1 χρησιμοποιούμε την κλάση ArrayList ως πρωταρχικό (raw type) και γενικευμένο τύπο. Στη συνάρτηση rawType χρησιμοποιείται ως πρωταρχικός τύπος. Σε αυτή τη μορφή της, ο τύπος των αντικειμένων που αποθηκεύει είναι Object. Καθώς η Object είναι γονική κλάση όλων των τύπων στην Java, το πραγματικό αντικείμενο μπορεί να είναι οποιασδήποτε κλάσης. Έτσι στην rawType αποθηκεύουμε έναν Integer, ένα String και ένα Object. Επομένως κατά την αποθήκευση των δεδομένων δεν μπορεί να γίνει έλεγχος του τύπου, καθώς όλες οι κλάσεις είναι αποδεκτές. Επιπλέον κατά την ανάκτηση πρέπει να γνωρίζουμε τι τύπο έχουμε αποθηκεύσει σε κάθε θέση του ArrayList ώστε να προβούμε στην κατάλληλη μετατροπή τύπου.

Αντίθετα, στη συνάρτηση generic δηλώνουμε ένα ArrayList που δέχεται αποκλειστικά String. Επομένως, υπάρχει η δυνατότητα πλήρους ελέγχου του τύπου κατά την αποθήκευση των δεδομένων, ενώ παράλληλα κατά την ανάκτησή τους είναι γνωστός ο τύπος τους, οπότε η μετατροπή τύπου είναι περιττή.

Ο όρος πρωταρχικός τύπος αναφέρεται σε γενικευμένο τύπο που χρησιμοποιείται χωρίς παράμετρο τύπου. Δεν ισχύει όμως για τύπους που δεν αποτελούν γενικεύσεις. Οι γενικεύσεις προστέθηκαν στην Java στην έκδοση 5. Πριν από αυτήν, ο μόνος τρόπος για να χρησιμοποιηθεί ένα ArrayList ήταν αυτός που επιδεικνύεται στη συνάρτηση rawType. Μετά την προσθήκη των γενικεύσεων, οι πρωταρχικοί τύποι διατηρήθηκαν μόνο για λόγους συμβατότητας με τους κώδικες που είχαν ήδη αναπτυχθεί, ωστόσο δεν συνίσταται η χρήση τους σε νέους κώδικες.

17.1 Ορισμός γενικευμένων κλάσεων

Μία γενικευμένη κλάση είναι στην ουσία μια κλάση παραμετροποιημένη προς έναν ή περισσότερους τύπους:

```
class OneTypeParam<T> {

    private T t;

    public OneTypeParam(T t) {
        this.t = t;
    }

    public void setT(T t) {
        this.t = t;
    }

    public T getT() {
        return t;
    }
}
```

Κώδικας 17.2 *Ορισμός γενίκευσης με μία παράμετρο τύπου*

Στον κώδικα 17.2 ορίζουμε μία γενικευμένη κλάση. Η T αποτελεί μεταβλητή τύπου. Κατά τον ορισμό της κλάσης, η T δεν αντιστοιχεί σε σαφή τύπο. Κατά τη δημιουργία όμως των αντικειμένων είμαστε υποχρεωμένοι να ορίσουμε σαφώς έναν τύπο όπως δείχνει ο παρακάτω κώδικας:

```
OneTypeParam<String> oTP = new OneTypeParam<>("Jim");
```

Το αντικείμενο oTP δημιουργείται αφού ο μεταγλωττιστής παραγάγει μία μη γενικευμένη κλάση στην οποία αντικαθιστά κάθε T με το String. Η διαδικασία αυτή αναφέρεται ως αντικατάσταση τύπου (type erasure). Αντίστοιχα, μία δήλωση του τύπου

```
OneTypeParam<Integer> oTP = new OneTypeParam<>(150);
```

θα ωθήσει τον μεταγλωττιστή να παράξει μία κλάση στην οποία κάθε T αντικαθίσταται από Integer.

Γενικά, για να δημιουργήσουμε ένα αντικείμενο τύπου OneTypeParameter<Integer>, θα πρέπει να καλέσουμε new OneTypeParameter<Integer>. Ωστόσο, όταν ο τύπος μπορεί να υπολογιστεί από τα συμφραζόμενα (context) όπως στα παραδείγματα που ήδη δώσαμε, τότε μπορούμε να χρησιμοποιήσουμε ένα

άδειο σύνολο από παραμέτρους τύπων. Το άδειο σύνολο εκφράζεται από τον τελεστή διαμαντιού (diamond operator) που συμβολίζεται ως <>. Η διαδικασία αυτόματης αναγνώρισης των παραμέτρων τύπων από τα συμφραζόμενα ονομάζεται εξόρυξη τύπου (type inference).
Ένας γενικευμένος τύπος μπορεί να παραμετροποιείται για περισσότερες από μία μεταβλητές τύπου όπως δείχνει ο κώδικας 17.3:

```
class TwoTypeParam<K, V> {

    private K key;
    private V value;

    public TwoTypeParam(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    public void setKey(K key) {
        this.key = key;
    }

    public void setValue(V value) {
        this.value = value;
    }
}
```

Κώδικας 17.3 *Ορισμός γενίκευσης με δύο παραμέτρους τύπου.*

Έχοντας τον κώδικα 17.3 στη διάθεσή μας, μπορούμε να ορίσουμε αντικείμενα ως εξής:

```
TwoTypeParam<String,Integer>tTP=new TwoTypeParam<>("Jim",120);
```

Επιπλέον, μια παράμετρος τύπου μπορεί να είναι και η ίδια μια γενίκευση όπως δείχνει η ακόλουθη δήλωση:

```
TwoTypeParam<Integer,OneTypeParam<String>>                t=new
TwoTypeParam<>(5,new OneTypeParam<>("Jim"));
```

Ας σημειωθεί πως στη θέση των μεταβλητών τύπου είναι αποδεκτές μόνο κλάσεις και όχι θεμελιώδεις τύποι.

17.2 Συναρτήσεις παραμετροποιημένες ως προς τύπο

Πέραν των κλάσεων είναι δυνατόν να παραμετροποιηθούν ως προς έναν ή περισσότερους τύπους και οι συναρτήσεις:

```
public class GenericFunctions {

    static <K, V> boolean equalId(TwoTypeParam<K, V> p1, TwoTypeParam<K, V> p2) {
        return p1.getKey().equals(p2.getKey())
                && p1.getValue().equals(p2.getValue());
    }

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

```
        TwoTypeParam<String, Integer> t1 = new TwoTypeParam<>("Jim", 100),
                t2 = new TwoTypeParam<>("Jim", 100),
                t3 = new TwoTypeParam<>("John", 200);
        System.out.println(equalId(t1, t2));
        System.out.println(equalId(t1, t3));
        TwoTypeParam<String, Double> t4=new TwoTypeParam<>("Jim", 100d);
    }
}
```

Κώδικας 17.4 *Η συνάρτηση equalId παραμετροποιημένη ως προς δύο τύπους*

Όπως φαίνεται στον κώδικα 17.4, το συντακτικό ορισμού συνάρτησης παραμετροποιημένης ως προς τύπο απαιτεί πριν τον τύπο επιστροφής της συνάρτησης να δηλωθεί το σύνολο των μεταβλητών τύπου. Στη συνέχεια οι μεταβλητές τύπου μπορούν να αξιοποιηθούν μέσα στο σώμα της συνάρτησης. Κατά την κλήση αντικαθίστανται όπως και στην περίπτωση των κλάσεων με μεταβλητές των πραγματικών τύπων. Προσέξτε όμως, η κλήση της equalId, όπως φαίνεται στον κώδικα που ακολουθεί, παράγει λάθος μεταγλώττισης.

```
TwoTypeParam<String, Double> t4=new TwoTypeParam<>("Jim",
100d);
System.out.println(equalId(t1,t4));
```

Πράγματι, η equalId διαθέτει δύο μεταβλητές παραμέτρου, την K και την V. Παράλληλα στη λίστα των παραμέτρων της οι δύο TwoTypeParam είναι και οι δύο <K,V>. Οι πραγματικές παράμετροι όμως που περνάμε είναι <String, Integer> και <String, Double>. Σε κάθε κλήση όμως της equal τόσο το K όσο και το V πρέπει να αναφέρονται σε συγκεκριμένο τύπο. Φυσικά, δεν θα υπήρχε πρόβλημα αν περνούσαμε δύο TwoTypeParam<String, Double>.

17.3 Παραμετροποιημένοι τύποι με όρια

Ας υποθέσουμε πως χρειαζόμαστε μία συνάρτηση που λαμβάνει ως παράμετρο μια λίστα από αντικείμενα Person ή μία λίστα από αντικείμενα Student και για κάθε ένα εμφανίζει το όνομά του, το επώνυμό του και την ηλικία του.
Ίσως, η πρώτη σκέψη μας να αποδίδεται από τον κώδικα 17.5.

```
    static void error(List<Person> table) {

        for (Person p : table) {
            System.out.println(p.getfName()+" "+p.getsName()+" "+p.getAge());
        }
    }
```

Κώδικας 17.5 *Εμφάνιση στοιχείων λίστας*

Πράγματι, ο κώδικας τρέχει σωστά με οποιαδήποτε λίστα τύπου List<Person>. Ωστόσο, ο κώδικας δεν μεταγλωττίζεται αν προσπαθήσουμε να περάσουμε ως πραγματική παράμετρο μία List<Student>. Παρότι η κλάση Student είναι παράγωγη της Person, η List<Student> δεν σχετίζεται ιεραρχικά με την List<Person>.

Τη λύση σε αυτό το πρόβλημα προσφέρουν οι παραμετροποιημένοι τύποι με όρια (bounded type parameters). Εκείνο που πραγματικά επιθυμούμε σε σχέση με μία τέτοια συνάρτηση είναι να δέχεται ως παράμετρο μια λίστα από αντικείμενα Person ή μία λίστα από αντικείμενα Student ή γενικότερα μία λίστα από αντικείμενα κλάσης παράγωγης της Person. Αυτό επιτυγχάνεται εύκολα, όπως δείχνει ο ακόλουθος κώδικας:

```
    static <T extends Person> void report(List<T> table) {

        for (Person p : table) {
            System.out.println(p.getfName()+" "+p.getsName()+" "+p.getAge());
        }
    }
```

Κώδικας 17.6 *Εμφάνιση στοιχείων γενικευμένης λίστας*

Η report τώρα δέχεται ως παράμετρο και List<Person> και List<Student>. Ας σημειωθεί πως στην List<Person> μπορεί να υπάρχουν αντικείμενα τύπου Student. Αντίθετα στην List<Student> δεν μπορεί να υπάρχουν αντικείμενα τύπου Person.

Επιπλέον, η λέξη extends εδώ έχει διαφοροποιημένη έννοια, καθώς μπορεί να ακολουθείται από κλάση ή διεπαφή. Στην περίπτωση που ακολουθείται από διεπαφή σημαίνει πως απαιτείται κλάση που υλοποιεί τη διεπαφή.

17.4 Η κληρονομικότητα στις γενικεύσεις

Η κληρονομικότητα στις γενικεύσεις συχνά αποτελεί ζήτημα που προκαλεί σύγχυση. Ας το δούμε με ένα παράδειγμα. Έστω ότι έχουμε μια συνάρτηση f:

```
static void f(Number k) {}
```

Οι κλήσεις

```
f(8);
f(8d)
```

είναι και οι δύο αποδεκτές. Τόσο η Integer όσο και η Double είναι παράγωγες της Number και όπως έχουμε μάθει αντικείμενο παράγωγης είναι αποδεκτό εκεί όπου αναμένεται αντικείμενο γονικής.
Παρόμοια, η συνάρτηση g

```
static <T extends Number> void g(T t) {}
```

μπορεί να κληθεί ως ακολούθως

```
g(8);
g(8d);
```

Η συνάρτηση

```
static void k(List<Number> l) {}
```

μπορεί να κληθεί ως

```
k(new ArrayList<Number>());
k(new LinkedList<Number>());
```

Όμως η κλήση

```
k(new ArrayList<Integer>());
```

δεν είναι σωστή.
Πράγματι η ArrayList<Number> είναι παράγωγη της List<Number>. Το ίδιο ισχύει και για την LinkedList<Number>. Ωστόσο η ArrayList<Integer> δεν είναι παράγωγη της List<Number>, παρότι η ArrayList είναι παράγωγη της List και η Integer παράγωγη της Number. Ίσως εκ πρώτης όψεως να είναι λίγο δύσκολο να το κατανοήσει κανείς, γίνεται όμως ευκολότερο αν ανακαλέσει πώς λειτουργεί ο μεταγλωττιστής με τις γενικεύσεις. Έχουμε πει πως όταν παράγει ένα αντικείμενο κάποιας γενίκευσης, αντικαθιστά τις μεταβλητές τύπου με την εξειδίκευση κατά τη δημιουργία των αντικειμένων. Για παράδειγμα, η ArrayList<Number> σε γενικές γραμμές έχει την ακόλουθη μορφή:

```
class ArrayListNumber {
```

```
        Number[] store;
        …
        …
    }
```

Αντίθετα, η ArrayList<Integer> έχει τη μορφή

```
    class ArrayListInteger {
        Integer[] store;
        …
        …
    }
```

Είναι προφανές πως οι δύο κλάσεις δεν συνδέονται με σχέσεις ιεραρχίας. Γενικότερα ισχύει πως αν έχουμε δύο κλάσεις A και B, ο τύπος MyClass<A> δεν σχετίζεται ιεραρχικά με τον τύπο MyClass<B>, ακόμη και αν η κλάση B είναι παράγωγη της A. Ο κοινός πρόγονος των MyClass<A> και MyClass<B> είναι η κλάση Object [1].

Γενικότερα, όταν μια γενίκευση κληρονομεί μία άλλη ή υλοποιεί μια γενικευμένη διεπαφή, η μεταξύ τους σχέση κληρονομικότητας διατηρείται μόνο για την ίδια παράμετρο τύπου. Για παράδειγμα, η ArrayList<E> υλοποιεί την List<E>. Επομένως, ένα ArrayList<Integer> είναι ένα List<Integer>, δεν είναι όμως ένα List<Number>, εξάλλου δεν είναι καν ArrayList<Number>.

17.5 Ο χαρακτήρας μπαλαντέρ

Ας υποθέσουμε τώρα πως χρειαζόμαστε μια συνάρτηση που να υπολογίζει το άθροισμα μιας λίστας αριθμών. Η συνάρτηση θα πρέπει να δουλεύει με όλες τις κλάσεις που υλοποιούν τη διεπαφή List αλλά και με όλους τους τύπους αριθμών, δηλαδή με όλες τις παράγωγες κλάσεις της Number. Με άλλα λόγια, θα μας εξυπηρετούσε μία συνάρτηση που μπορεί να λάβει ως παράμετρο και ArrayList<Double> και LinkedList<Integer> αλλά και όλους τους υπόλοιπους συνδυασμούς λιστών και αριθμών.

Τη δυνατότητα αυτή μας παρέχει ο χαρακτήρας μπαλαντέρ (WildCard) που αναπαρίσταται με το αγγλικό ερωτηματικό:

```
static double sum(List<? extends Number> l) {
    double sum = 0d;
    for (int i = 0; i < l.size(); i++) {
        sum += l.get(i).doubleValue();
    }
    return sum;
}
```

Κώδικας 17.7 *Αξιοποίηση του χαρακτήρα μπαλαντέρ με άνω όριο*

Η έκφραση List<? extends Number> στη λίστα παραμέτρων της sum σημαίνει πως η συνάρτηση δέχεται ως παραμέτρους αντικείμενο οποιασδήποτε κλάσης υλοποιεί την List, ενώ τα στοιχεία του είναι Number ή παράγωγης της Number. Έτσι, οι κλήσεις στον κώδικα 17.8 είναι όλες σωστές:

```
public static void main(String[] args) {
    List<Double> ld=new ArrayList<>();
    List<Integer> li=new ArrayList<>();
    List<Number> ln=new LinkedList<>();
    for (int i=0; i<10; i++) {
        ld.add((double)i);
        li.add(i);
        ln.add(i);
    }
    System.out.println(sum1(ld));
    System.out.println(sum1(li));
    System.out.println(sum1(ln));
```

```
    }
```

Κώδικας 17.8 *Κλήσεις της γενικευμένης συνάρτησης sum*

Ας σημειωθεί πως εναλλακτική προσέγγιση σε αυτό το πρόβλημα αποτελεί μία συνάρτηση με την ακόλουθη διεπαφή:

```
static <T extends Number> double sum1(List<T> l)
```

Στο παράδειγμα που αναπτύχθηκε στους κώδικες 17.7 και 17.8 περιορίσαμε τα στοιχεία της λίστας ώστε τα στοιχεία της να είναι Number ή παράγωγες κλάσεις. Με άλλα λόγια, δώσαμε το άνω όριο (upper bound) των κλάσεων που είναι αποδεκτές ως τύπος της λίστας. Αυτό ήταν αναγκαίο, καθώς μέσα στο σώμα της sum εκτελούμε προσθέσεις με αυτά τα στοιχεία. Αν όμως θέλαμε μια επεξεργασία που δεν απαιτεί συγκεκριμένο τύπο θα μπορούσαμε να χρησιμοποιήσουμε τον χαρακτήρα μπαλαντέρ χωρίς όρια όπως δείχνει ο κώδικας 17.9:

```
    static void prt(List<?> l) {
        for (int i=0; i<l.size(); i++) {
            System.out.println(l.get(i));
        }
    }
```

Κώδικας 17.9 *Αξιοποίηση του χαρακτήρα μπαλαντέρ χωρίς όρια*

Ας υποθέσουμε τώρα πως χρειαζόμαστε μία συνάρτηση που να εισάγει αντικείμενα τύπου Integer σε μία λίστα. Εκ πρώτης όψεως φαίνεται εύκολη απαίτηση, εκτός αν λάβουμε υπόψη ότι σε αντικείμενα Integer μπορούμε να αναφερόμαστε με μεταβλητές τύπου Number αλλά και με μεταβλητές τύπου Object, δηλαδή με μεταβλητές που ο τύπος τους είναι Integer ή κάποιος πρόγονος της Integer. Επομένως, η συνάρτησή μας θα πρέπει να ανταποκρίνεται σωστά με List<Integer>, με List<Number> και με List<Object>.
Μια τέτοια απαίτηση μπορούμε να υλοποιήσουμε με χρήση μπαλαντέρ με κάτω όριο (lower bound).

```
import java.util.ArrayList;
import java.util.List;

public class WCLowBounds {

    static void insert(List<? super Integer> l) {
        Integer i = 3;
        l.add(i);
    }

    public static void main(String[] args) {
        ArrayList<Integer> li = new ArrayList<>();
        ArrayList<Number> ln = new ArrayList<>();
        ArrayList<Object> lo = new ArrayList<>();
        insert(li);
        insert(ln);
        insert(lo);
    }
}
```

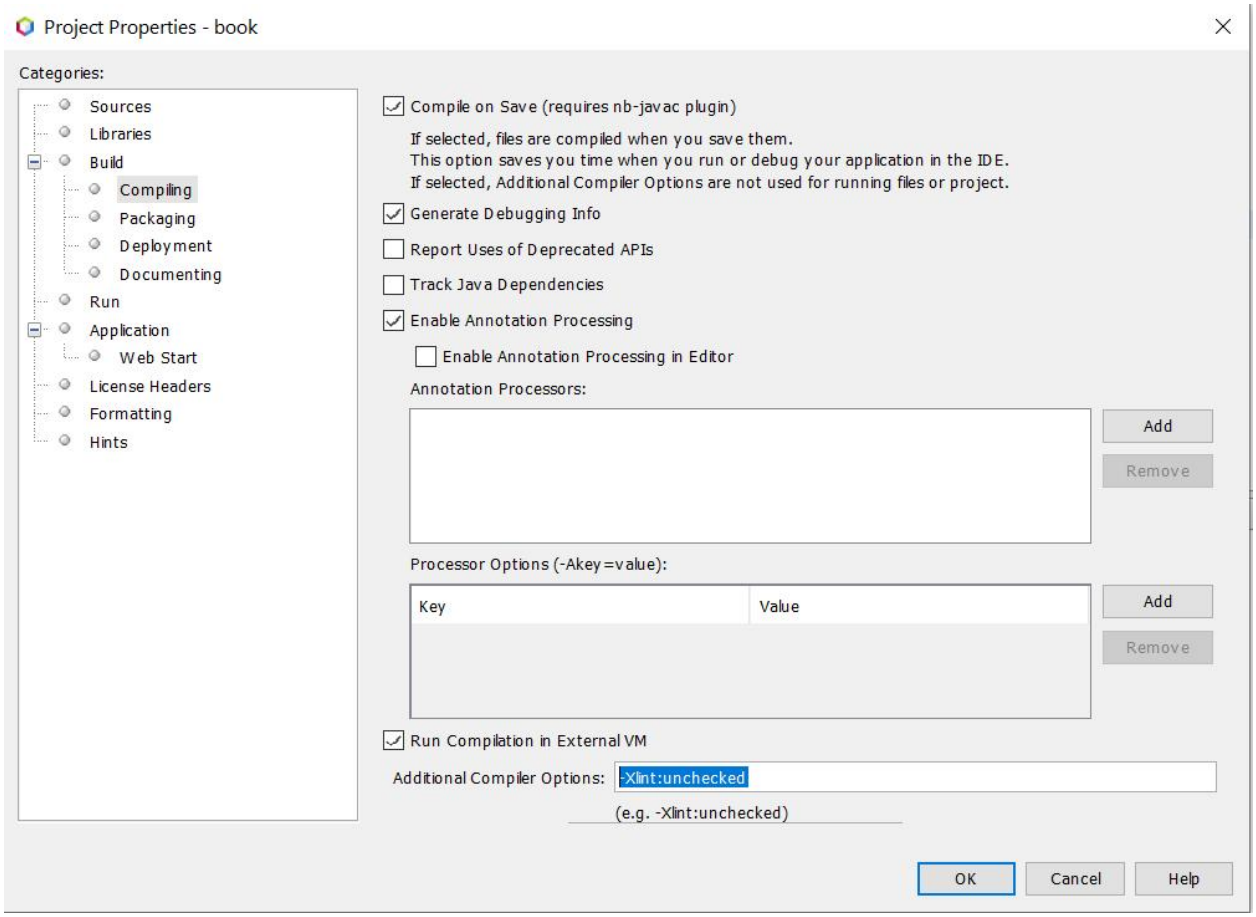
Κώδικας 17.10 *Αξιοποίηση του χαρακτήρα μπαλαντέρ με κάτω όριο*

Στον κώδικα 17.10 η συνάρτηση insert δηλώνει την παράμετρό της με χρήση μπαλαντέρ με κάτω όριο. Όπως φαίνεται στον κώδικα, η δήλωση γίνεται με τη δεσμευμένη λέξη super. Η insert μπορεί να λάβει ως παράμετρο οποιαδήποτε λίστα από Integers, Numbers ή Objects. Στη λίστα όμως επιτρέπεται μόνο η εισαγωγή Integer.

Ανακεφαλαιώνοντας, επισημαίνουμε τα εξής: Η List<Number> δεν είναι γονική της List<Integer>. Και οι δύο κλάσεις είναι παράγωγες της List<?>. Όμως η List<? extends Number> είναι γονική της List<? extends Integer>.

17.6 Μη Ασφαλείς Λειτουργίες

Σε μερικές περιπτώσεις έχουμε χρησιμοποιήσει μη ασφαλείς λειτουργίες (unsafe operations). Ο μεταγλωττιστής της Java επιτρέπει τις μη ασφαλείς λειτουργίες, καθώς σε πολλές περιπτώσεις είναι αναγκαίες. Εξ ορισμού δεν παράγει καν προειδοποιήσεις. Αν θέλετε να δείτε τις μη ασφαλείς λειτουργίες του κώδικά σας θα πρέπει να μεταγλωττίσετε με την παράμετρο -Xlint:unchecked. Μεταβείτε στα Properties του project, επιλέξτε Compiling και ενημερώστε τις επιπρόσθετες επιλογές μεταγλώττισης (Additional Compiler Options) όπως δείχνει η εικόνα 17.1:



Εικόνα 17.1 *Επιπρόσθετες επιλογές μεταγλώττισης*

Τι είναι όμως ακριβώς οι μη ασφαλείς λειτουργίες; Θα το εξηγήσουμε με ένα παράδειγμα. Έστω ότι θέλουμε να υλοποιήσουμε μία γενικευμένη συνάρτηση ταξινόμησης πινάκων χωρίς να χρησιμοποιήσουμε την sort της Arrays. Προκειμένου να ταξινομηθούν τα στοιχεία ενός πίνακα θα πρέπει να είναι συγκρίσιμα μεταξύ τους. Οι κλάσεις που υλοποιούν τη διεπαφή Comparable<T> παρέχουν τη δυνατότητα σύγκρισης των αντικειμένων τους. Πολλές προκαθορισμένες κλάσεις της Java υλοποιούν την Comparable<T>. Αυτή καθορίζει τη φυσική σειρά (natural order) των αντικειμένων. Μπορείτε να βρείτε περισσότερες πληροφορίες στην τεκμηρίωση της Java. Πάντως, η συνάρτησή μας θα πρέπει να λαμβάνει πίνακες από αντικείμενα Comparable, όπως δείχνει ο κώδικας 17.11:

```
    private static <E> void swap(E[] a, int i, int j) {
        if (i != j) {
            E temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }

    public static <E extends Comparable<E>> void sort(E[] a) {
```

```
        for (int i = 0; i < a.length - 1; i++) {
            int smallest = i;
            for (int j = i + 1; j < a.length; j++) {
                if (a[j].compareTo(a[smallest]) <= 0) {
                    smallest = j;
                }
            }
            swap(a, i, smallest);
        }
    }
```

Κώδικας 17.11 *Γενικευμένη έκδοση της sort πινάκων*

Ας υποθέσουμε πως έχουμε στη διάθεσή μας και την κλάση Tmp του κώδικα 17.12.

```
class Tmp implements Comparable<Tmp> {

    private final String id;
    private final String description;

    public Tmp(String id, String description) {
        this.id = id;
        this.description = description;
    }

    @Override
    public int compareTo(Tmp o) {
        return id.compareTo(o.id);
    }

    @Override
    public String toString() {
        return id;
    }
}
```

Κώδικας 17.12 *Η Tmp που υλοποιεί την διεπαφή Comparable.*

Σε αυτήν την περίπτωση μπορούμε να δημιουργήσουμε πίνακες ως εξής:

```
Integer[] ints = new Integer[10];
Double[] doubles = new Double[10];
Tmp[] tmps = new Tmp[10];
```

Αν δώσουμε τιμές στα στοιχεία σε κάθε έναν από τους πίνακες ints, doubles και tmps και στη συνέχεια καλέσουμε την sort για κάθε πίνακα, θα διαπιστώσουμε ότι όλοι οι πίνακες ταξινομούνται σωστά χωρίς κανένα πρόβλημα. Θεωρήστε τώρα τον πίνακα

```
Comparable[] l=new Comparable[3];
```

που περιλαμβάνει στοιχεία ως εξής:

```
l[0]=10;
l[1]=2d;
l[2]=new Tmp(Integer.toString(1), Integer.toString(2));
```

και την κλήση

```
sort(tmps);
```

Ο κώδικας θα πέσει με java.lang.ClassCastException. Η κλήση sort(tmps) συνιστά μη ασφαλή κλήση, διότι αν προσέξετε τη διεπαφή της sort θα δείτε πως απαιτεί πίνακα στοιχείων τύπου E ο οποίος να είναι Comparable. Ωστόσο είναι συγκεκριμένος τύπος. Αντίθετα στον πίνακα l περιλαμβάνονται μεν Comparables διαφορετικών ωστόσο τύπων. Έτσι, όταν μέσα στην sort η compareTo επιχειρεί να συγκρίνει ένα αντικείμενο τύπου Integer με ένα τύπου Double ή τύπου Tmp, προσπαθεί να μετατρέψει τον τύπο Double ή Tmp σε Integer, αποτυγχάνει και πέφτει με ClassCastException. Αν ωστόσο όλα τα στοιχεία του πίνακα l αφορούν την ίδια κλάση δεν υπάρχει πρόβλημα παρότι η κλήση συνεχίζει να χαρακτηρίζεται ως μη ασφαλής. Εναπόκειται επομένως στην ευθύνη του κώδικα-πελάτη να καλέσει την sort με σωστό τρόπο.
Σε γενικές γραμμές θα πρέπει να αποφεύγονται οι μη ασφαλείς λειτουργίες. Σε πολλές περιπτώσεις όμως είναι αναγκαίες. Για παράδειγμα, στον κώδικα 15.8, η συνάρτηση loadCars περιέχει τη γραμμή

```
ArrayList<Car> loaded = (ArrayList<Car>) in.readObject();
```

Η readObject επιστρέφει αντικείμενο τύπου Object το οποίο μετατρέπουμε σε ArrayList<Car>. Ο μεταγλωττιστής δεν μπορεί να γνωρίζει αν η μετατροπή αυτή είναι σωστή. Με αυτήν την έννοια, η μετατροπή χαρακτηρίζεται ως μη ασφαλής. Ωστόσο, είναι αναγκαία εφόσον είναι ο μόνος τρόπος για να φορτώσουμε στη μνήμη ένα σειριοποιημένο ArrayList<Car>. Πάλι εναπόκειται στην ευθύνη του κώδικα πελάτη να κάνει τη σωστή μετατροπή, διαφορετικά θα λάβει ClassCastExeption.
Ένα άλλο παράδειγμα υπάρχει στον κώδικα 12.20. Προσέξτε πως στην αρχή της κλάσης MyHashMap υπάρχει η γραμμή

```
private final ArrayList<Person>[] store = new ArrayList[SIZE];
```

Η Java δεν επιτρέπει τη δημιουργία γενικευμένων πινάκων. Ο τύπος του πίνακα store είναι ArrayList<Person>, ωστόσο, ο πραγματικός πίνακας που δημιουργείται είναι ArrayList (raw type). Προσέξτε ότι κατά τη δημιουργία δεν χρησιμοποιήθηκε ο τελεστής διαμαντιού, διαφορετικά η γραμμή θα είχε ως εξής:

```
private final ArrayList<Person>[] store = new
ArrayList<>[SIZE];
```

Ωστόσο, η τελευταία αυτή γραμμή δεν περνάει μεταγλώττιση. Έτσι, όταν χρειαζόμαστε έναν πίνακα στον οποίο θα αποθηκεύουμε γενικεύσεις, στην πράξη δημιουργούμε έναν πίνακα από Object. Καθώς η Object είναι γονική όλων των κλάσεων, είναι και γονική κάθε γενίκευσης. Με άλλα λόγια σε μία αναφορά τύπου Object μπορούμε να εκχωρήσουμε οποιαδήποτε γενίκευση. Επομένως, οι πίνακες στους οποίους αποθηκεύουμε γενικεύσεις μάς εξαναγκάζουν να χρησιμοποιούμε μη ασφαλείς λειτουργίες. Ας σημειωθεί πως η ίδια η προκαθορισμένη κλάση Collections δεν μεταγλωττίζεται χωρίς προειδοποιήσεις (warnings) για μη ασφαλείς λειτουργίες.

17.7 Λυμένες Ασκήσεις

17.7.1 MyHashMap<K,V>

Να υλοποιηθεί η γενίκευση MyHashMap<K,V> χωρίς να χρησιμοποιηθεί η προκαθορισμένη γενίκευση HashMap<K,V>. Ο εσωτερικός αποθηκευτικός χώρος της MyHashMap θα πρέπει να είναι ένας πίνακας σε κάθε θέση του οποίου θα τοποθετείται ένα ArrayList στο οποίο θα εισάγονται οι συσχετίσεις K, V ανάλογα με τον κωδικό κατακερματισμού τους. Ο εσωτερικός πίνακας θα δημιουργείται με αρχικό μέγεθος 16. Επιπλέον, θα οριστεί ένας συντελεστής φόρτωσης. Όταν οι κατειλημμένες θέσεις του εσωτερικού πίνακα υπερβαίνουν το μέγεθος του πίνακα επί τον συντελεστή φόρτωσης, το μέγεθος του εσωτερικού πίνακα θα πρέπει να διπλασιάζεται και οι συσχετίσεις να επανατοποθετούνται, δηλαδή να γίνεται επανακατακερματισμός (rehashing). Σε ένα MyHashMap δεν επιτρέπεται η εισαγωγή συσχετίσεων με το ίδιο κλειδί. Η MyHashMap θα πρέπει να υποστηρίζει τις ακόλουθες μεθόδους:

```
public void clear()
```

Διαγράφει όλες τις συσχετίσεις από αυτό το αντικείμενο

`public V put(K key, V value)`

Εισάγει τη συσχέτιση μεταξύ key και value. Αν η συσχέτιση υπάρχει ήδη, τότε την ενημερώνει και επιστρέφει την τιμή V που βρέθηκε στο MyHashMap. Αν δεν υπάρχει, την εισάγει και επιστρέφει null.

`public V get(K key)`

Επιστρέφει την τιμή V που σχετίζεται με αυτό το κλειδί. Αν το κλειδί δεν βρεθεί, επιστρέφει null.

`public V remove(K key)`

Διαγράφει από αυτό το MyHashMap τη συσχέτιση με κλειδί key. Αν η συσχέτιση δεν βρεθεί, επιστρέφει null αλλιώς τη συσχέτιση που διεγράφη.

`public boolean isEmpty()`

Επιστρέφει true αν καμία συσχέτιση δεν υπάρχει σε αυτό το MyHshMap.

Προσθέστε main στην οποία δημιουργήστε ένα αντικείμενο MyHashMap<String, String>. Εισάγεται μερικές συσχετίσεις. Ρυθμίστε την αρχική χωρητικότητα του εσωτερικού πίνακα και τον συντελεστή φόρτωσης έτσι ώστε με την εισαγωγή των συσχετίσεων να προκληθεί επανακατακερματισμός. Διαβάστε τις συσχετίσεις με την get. Στη συνέχεια, διαγράψτε τις συσχετίσεις μια προς μία με την remove. Μετά από κάθε διαγραφή, καλέστε την isEmpty. Ελέγξτε ώστε όλα τα αποτελέσματα να είναι τα αναμενόμενα.

Λύση

```
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class MyHashMap<K, V> {

    public class Entry<K, V> {

        private final K key;
        private V value;

        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public K getKey() {
            return key;
        }

        public V getValue() {
            return value;
        }

        public void setValue(V value) {
            this.value = value;
        }

        @Override
        public int hashCode() {
            int hash = 7;
            hash = 97 * hash + Objects.hashCode(this.key);
            return hash;
        }

        @Override
        public boolean equals(Object obj) {
```

```
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Entry<K, V> other = (Entry<K, V>) obj;
        return Objects.equals(this.key, other.key);
    }
}

private int find(K key, List<Entry<K, V>> list) {
    for (int i = 0; i < list.size(); i++) {

        Entry<K, V> entry = list.get(i);
        if (entry.key.equals(key)) {
            return i;
        }
    }
    return -1;
}

private int size = 0;
private static final double LOADFACTOR = 0.75;
private int noOfLists = 0;
private final int INITIALCAPACITY = 3;
private int capacity = INITIALCAPACITY;
private ArrayList<Entry<K,V>>[] store = new ArrayList[INITIALCAPACITY];

public void clear() {
    store = new ArrayList[INITIALCAPACITY];
    size = 0;
    capacity = INITIALCAPACITY;
    noOfLists = 0;
}

private void rehash() {
    int newCapacity = 2 * capacity;
    int newEntries = 0;
    ArrayList<Entry<K,V>>[] local = new ArrayList[newCapacity];
    for (ArrayList<Entry<K,V>> list : store) {

        if (list != null) {
            for (Entry<K,V> entry : list) {
                                int hashCode = entry.key.hashCode();
                int idx = hashCode % newCapacity;
                if (local[idx] == null) {
                    local[idx] = new ArrayList<Entry<K, V>>();
                    newEntries++;
                }
                local[idx].add(entry);
            }
        }
    }
    store = local;
    noOfLists = newEntries;
    capacity = newCapacity;
}
```

```
public V put(K key, V value) {
    if (noOfLists > LOADFACTOR * capacity) {
        rehash();
    }
    int idx = key.hashCode() % store.length;
    if (store[idx] == null) {
        store[idx] = new ArrayList<Entry<K, V>>();
        noOfLists++;
    }
    Entry<K, V> entry = new Entry<>(key, value);
    int location = store[idx].indexOf(entry);
    Entry<K, V> oldEntry;
    if (location < 0) {
        store[idx].add(entry);
        size++;
        return null;
    } else {
        oldEntry = store[idx].get(location);
        store[idx].set(location, entry);
    }
    return oldEntry.getValue();
}

public V get(K key) {
    int idx = key.hashCode() % store.length;

    int location = find(key, store[idx]);
    if (location < 0) {
        return null;
    }
    return store[idx].get(location).getValue();
}

public V remove(K key) {
    int idx = key.hashCode() % store.length;
    int location = find(key, store[idx]);
    if (location < 0) {
        return null;
    }
    size--;
    V rVal = store[idx].remove(location).getValue();
    if (store[idx].isEmpty()) {
        noOfLists--;
        store[idx] = null;
    }
    return rVal;
}

public boolean isEmpty() {
    return size==0;
}

public static void main(String[] args) {
    MyHashMap<String, String> hM = new MyHashMap<>();
    hM.put("key1", "value1");
    hM.put("key2", "value2");
    hM.put("key3", "value3");
    hM.put("key4", "value4");
    hM.put("key5", "value5");
    hM.put("key6", "value6");
```

```
            hM.put("key7", "value7");
            hM.put("key8", "value8");
            System.out.println(hM.get("key1"));
            System.out.println(hM.get("key2"));
            System.out.println(hM.get("key3"));
            System.out.println(hM.get("key4"));
            System.out.println(hM.get("key5"));
            System.out.println(hM.get("key6"));
            System.out.println(hM.get("key7"));
            System.out.println(hM.get("key8"));

            System.out.println(hM.remove("key1"));
            System.out.println(hM.isEmpty());
            System.out.println(hM.remove("key2"));
            System.out.println(hM.isEmpty());
            System.out.println(hM.remove("key3"));
            System.out.println(hM.isEmpty());
            System.out.println(hM.remove("key4"));
            System.out.println(hM.isEmpty());
            System.out.println(hM.remove("key5"));
            System.out.println(hM.isEmpty());
            System.out.println(hM.remove("key6"));
            System.out.println(hM.isEmpty());
            System.out.println(hM.remove("key7"));
            System.out.println(hM.isEmpty());
            System.out.println(hM.remove("key8"));
            System.out.println(hM.isEmpty());
        }
    }
```

Κώδικας 17.13 *Η κλάση MyHashMap<K,V>*

Η κλάση MyHashMap<K,V> στον κώδικα 17.11 ορίζει την εσωτερική κλάση Entry<K,V>. Τα αντικείμενα Entry<K,V> συνιστούν τις συσχετίσεις που εισάγονται στο MyHashMap. Από τη στιγμή που δημιουργείται μία τέτοια συσχέτιση, το κλειδί της δεν μπορεί να μεταβληθεί. Για αυτόν τον λόγο το μέλος key δηλώθηκε ως final και δεν παρέχεται αντίστοιχος ρυθμιστής. Καθώς στο MyHashMap δεν επιτρέπεται η εισαγωγή συσχετίσεων με το ίδιο κλειδί, η equals θεωρεί συσχετίσεις με το ίδιο κλειδί ως ίσες, οπότε και η hashCode βασίζεται αποκλειστικά στο κλειδί.

Η κλάση διαθέτει τα μέλη δεδομένα, size που διατηρεί τον αριθμό των καταχωρημένων συσχετίσεων, LOADFACTOR που εκφράζει τον συντελεστή φόρτωσης, noOfLists που διατηρεί τον αριθμό των ενεργών λιστών, INITIALCAPACITY που ρυθμίζει την αρχική χωρητικότητα του εσωτερικού πίνακα, capacity που διατηρεί την τρέχουσα χωρητικότητα του εσωτερικού πίνακα και τον εσωτερικό πίνακα store. Προσέξτε πως ο store ως προς τον τύπο είναι πίνακας από ArrayList<Entry<K,V>>. Ωστόσο ο πίνακας που δημιουργείται είναι πρωταρχικός τύπος, καθώς η δημιουργία πινάκων γενικεύσεων δεν υποστηρίζεται. Επομένως, η δημιουργία του store είναι αναγκαστικά μη ασφαλής λειτουργία. Ωστόσο, η κλάση οργανώνεται κατά τρόπο που δεν υποχρεώνει τον χρήστη της σε μη ασφαλείς λειτουργίες. Αντίθετα, όλες οι μη ασφαλείς λειτουργίες είναι εσωτερικές στην κλάση.

Η find είναι μια ιδιωτική συνάρτηση που ψάχνει σε μία λίστα να βρει τη συσχέτιση με κλειδί key. Αν την βρει επιστρέφει τη θέση της στη λίστα, αν όχι επιστρέφει -1.

Η clear δημιουργεί νέο εσωτερικό χώρο και αρχικοποιεί κατάλληλα τις μεταβλητές size, capacity, και noOfLists.

Η put τοποθετεί μία συσχέτιση σε αυτό το MyHashMap. Η πρώτη ενέργεια της put είναι να ελέγξει τον αριθμό των ενεργών λιστών, δηλαδή τον αριθμό των κατειλημμένων θέσεων του πίνακα store. Αν αυτός υπερβαίνει το γινόμενο LOADFACTOR*capacity, τότε καλεί την rehash για να μεγεθύνει τον εσωτερικό αποθηκευτικό χώρο και να επανατοποθετήσει τις συσχετίσεις. Στη συνέχεια εντοπίζει το κλειδί κατακερματισμού της key και υπολογίζει τη θέση του store στην οποία πρέπει να αποθηκευτεί η συσχέτιση. Αν σε εκείνη τη θέση του πίνακα δεν υπάρχει λίστα, τη δημιουργεί και αυξάνει κατάλληλα την noOfLists. Μετά δημιουργεί τη συσχέτιση ως Entry, ελέγχει αν υπάρχει ήδη στην κατάλληλη λίστα και αν υπάρχει την

ενημερώνει και επιστρέφει την παλιά τιμή, αλλιώς την εισάγει και επιστρέφει null. Στην περίπτωση που την εισάγει ενημερώνει και τη μεταβλητή size.

Η get επιστρέφει την τιμή που συσχετίζεται με το κλειδί που λαμβάνει ως παράμετρο. Αν δεν υπάρχει συσχέτιση για αυτό το κλειδί, επιστρέφει null.

Η remove διαγράφει τη συσχέτιση στην οποία αναφέρεται το κλειδί που λαμβάνει ως παράμετρο και μειώνει το size κατά 1, εφόσον βρεθεί το κλειδί. Αν με τη διαγραφή της συσχέτισης η αντίστοιχη λίστα αδειάσει, τότε θέτει τη θέση της στον πίνακα στο null και μειώνει την noOfLists κατά 1.

Η isEmpty βασίζεται στη μεταβλητή empty που ενημερώνεται με κάθε εισαγωγή και διαγραφή συσχετίσεων από αυτό το MyHashMap.

Τέλος, η rehash δημιουργεί έναν νέο πίνακα, επανατοποθετεί τα στοιχεία του store στο νέο πίνακα και ανακατευθύνει τη μεταβλητή store ώστε να αναφέρεται στον νέο πίνακα. Ο παλαιότερος πίνακας μένει χωρίς αναφορά και είναι διαθέσιμος για διαγραφή από τον συλλέκτη απορριμμάτων.

17.7.2 Γενικευμένη συνάρτηση υπολογισμού αθροίσματος

Να υλοποιηθεί συνάρτηση sum που λαμβάνει ως παράμετρο λίστα ενός οποιουδήποτε αριθμητικού τύπου και επιστρέφει το άθροισμα των στοιχείων της λίστας ως double. Προσθέστε main στην οποία ελέγξτε την sum με ArrayList<Integer> και LinkedList<Double>.

Λύση

Η τυπική παράμετρος της συνάρτησης θα πρέπει να είναι του τύπου List<? extends Number>. Με αυτήν την παράμετρο, η συνάρτηση μπορεί να δεχτεί οποιαδήποτε γενίκευση εφόσον υλοποιεί τη διεπαφή List. Επιπλέον, η λίστα μπορεί να εξειδικεύεται ως προς οποιονδήποτε τύπο κληρονομεί την αφηρημένη κλάση Number. Στην περίπτωση που σκεφτήκατε να ορίσετε την παράμετρο ως List<Number>, προσέξτε το εξής: Αυτή η παράμετρος επιτρέπει μόνο λίστες από Number. Βεβαίως μία λίστα από Number μπορεί να περιέχει ως στοιχεία αντικείμενα των παράγωγων κλάσεων. Ωστόσο, δεν είναι δυνατό να περάσουμε στη θέση της μία λίστα από Integer ή μία λίστα από Double.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Calculator {

    public static double sum(List<? extends Number> list) {
        double sum = 0;
        Iterator<? extends Number> it = list.iterator();
        while (it.hasNext()) {
            sum += it.next().doubleValue();
        }
        return sum;
    }

    public static double product(List<? extends Number> list) {
        double product = 1;
        Iterator<? extends Number> it = list.iterator();
        while (it.hasNext()) {
            product *= it.next().doubleValue();
        }
        return product;
    }

    public static void main(String[] args) {
        List<Integer> li = new ArrayList<>();
        List<Double> ld = new LinkedList<>();
        List<Number> ln=new ArrayList<>();
```

```
            li.add(5);
            li.add(6);
            ld.add(5.2);
            ld.add(6.5);
            ln.add(5);
            ln.add(5.5);
            System.out.println(sum(li));
            System.out.println(sum(ld));
            System.out.println(sum(ln));
        }
    }
```

Κώδικας 17.14 *Η συνάρτηση sum(List<? extends Number>*

Από τη στιγμή που έχουμε προσδιορίσει σωστά τον τύπο της παραμέτρου, η υλοποίηση της συνάρτησης είναι απλή. Χρησιμοποιώντας τον Iterator που παίρνουμε από τη λίστα, διασχίζουμε όλα τα στοιχεία της και προσθέτουμε την πραγματική τιμή τους στον αθροιστή sum τον οποίο και επιστρέφουμε.

17.7.3 Κατασκευή πινάκων

Να υλοποιηθεί στατική συνάρτηση mkA με λίστα παραμέτρων μεταβλητού τύπου του παραμετροποιημένου τύπου T. Η συνάρτηση επιστρέφει έναν πίνακα τύπου T που περιέχει τα αντικείμενα που περνούν ως πραγματικές παράμετροι κατά την κλήση της.

Να προστεθεί ο ακόλουθος κώδικας στην main:

```
public static void main(String[] args) {
        String[] s = mkA("Jack", "John", "Mary");
        Integer[] k = mkA(1, 2, 3, 4, 5);
        for (int i = 0; i < s.length; i++) {
            System.out.println(s[i]);
            System.out.println(k[i]);
        }
    }
```

Λύση

Πρόκειται για πολύ απλή συνάρτηση. Το μόνο κρίσιμο σημείο είναι η σωστή ανάπτυξη της διεπαφής της συνάρτησης. Κατά τα λοιπά, η συνάρτηση απλώς επιστρέφει την παράμετρό της η οποία είναι πίνακας τύπου T και περιέχει όλες τις πραγματικές παραμέτρους.

```
public static <T> T[] mkA(T... a) {
    return a;
}
```

17.8 Ασκήσεις προς Λύση

17.8.1 Γενικευμένη δομή LIFO

Να υλοποιηθεί η γενικευμένη κλάση MyStack<E> που αναπαριστά μια δομή LIFO χωρίς να χρησιμοποιηθεί η προκαθορισμένη Stack<E>. Η κλάση πρέπει να υποστηρίζει τις λειτουργίες empty που επιστρέφει true αν αυτό το stack είναι άδειο, την push(E item) που εισάγει το item στην κορυφή της στοίβας και την pop που διαγράφει και επιστρέφει το item από την κορυφή της στοίβας.

17.8.2 Γενικευμένη δομή FIFO

Να υλοποιηθεί η γενικευμένη κλάση Queue<E> που αναπαριστά μια δομή FIFO. Η κλάση πρέπει να υποστηρίζει τις λειτουργίες empty που επιστρέφει true αν αυτή η ουρά είναι άδεια, την add(E item) που

εισάγει το item στο τέλος της ουράς και την delete που διαγράφει και επιστρέφει το στοιχείο στην αρχή της ουράς.

17.8.3 Γενικευμένη συνάρτηση υπολογισμού γινομένου

Να υλοποιηθεί συνάρτηση product που λαμβάνει ως παράμετρο λίστα ενός οποιουδήποτε αριθμητικού τύπου και επιστρέφει το γινόμενο των στοιχείων της λίστας ως double. Προσθέστε main στην οποία ελέγξτε την product με ArrayList<Integer> και LinkedList<Double>.

Βιβλιογραφία

[1] “Generics, Inheritance, and Subtypes (The Java™ Tutorials > Learning the Java Language > Generics (Updated)).” https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html (accessed Nov. 19, 2021).

Κεφάλαιο 18

Σύνοψη
Στην ενότητα αυτή παρουσιάζονται οι εκφράσεις λάμδα μαζί με τις λειτουργικές διεπαφές, οι αναφορές μεθόδων καθώς και η απομνημόνευση, μία τεχνική που περιορίζει των αριθμό των κλήσεων στις αναδρομικές συναρτήσεις και κάνει τον κώδικα πιο αποτελεσματικό.

Προαπαιτούμενη γνώση
Η ενσωμάτωση, η κληρονομικότητα και ο πολυμορφισμός στο αντικειμενοστρεφές μοντέλο με Java, οι αφηρημένες κλάσεις καθώς και οι βασικές προκαθορισμένες κλάσεις, δηλαδή οι String, StringBuilder, ArrayList, LinkedList, Stack, PriorityQueue, HashSet, HashMap, HashTree, LocalDate, Localtime, Period και Duration. Οι διεπαφές στην Java και οι γενικεύσεις.

Λέξεις κλειδιά
Λειτουργικές διεπαφές (functional interfaces), Εκφράσεις λάμδα (Lamda expressions), Αναφορές μεθόδων (method references), Απομνημόνευση (memoization).

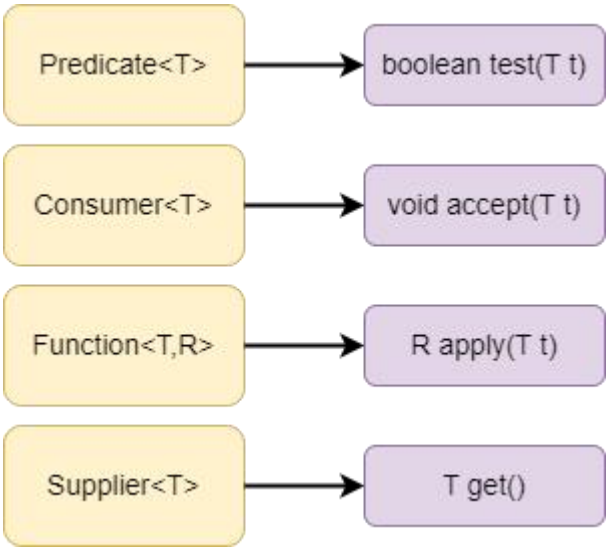
18 Ειδικά θέματα

18.1 Εκφράσεις Λάμδα

Συχνά χρειάζεται να περάσουμε ως παράμετρο σε μία συνάρτηση ένα αντικείμενο. Σε κάποιες περιπτώσεις είναι αντικείμενο μιας κλάσης που υλοποιεί μία διεπαφή. Σύμφωνα με όσα έχουμε ήδη μάθει, έχουμε δύο δυνατότητες. Να ορίσουμε την κλάση που υλοποιεί τη διεπαφή και στη συνέχεια το αντικείμενο ή να δημιουργήσουμε ένα αντικείμενο ανώνυμης κλάσης. Όμως αρκετά συχνά η διεπαφή έχει μία μόνο αφηρημένη μέθοδο. Επομένως, σε αυτές τις περιπτώσεις, το μόνο που είναι πράγματι αναγκαίο να κάνουμε είναι να ορίσουμε τον κώδικα της αφηρημένης μεθόδου. Θα ήταν μεγάλη ευκολία αν μπορούσαμε να περάσουμε ως παράμετρο κατευθείαν τον κώδικα που υλοποιεί την αφηρημένη μέθοδο και αποφεύγαμε τη δήλωση επώνυμης ή και ανώνυμης κλάσης. Αυτήν ακριβώς τη λειτουργικότητα μας παρέχουν οι εκφράσεις Λάμδα (Lamda expressions) που προστέθηκαν στην έκδοση 8 της Java.

18.1.1 Λειτουργικές διεπαφές

Λειτουργικές διεπαφές (functional interfaces) είναι αυτές που διαθέτουν μία μόνο αφηρημένη μέθοδο [1]. Βεβαίως, μια λειτουργική διεπαφή μπορεί να περιλαμβάνει πολλές στατικές ή προκαθορισμένες μεθόδους. Η Java διαθέτει ένα μεγάλο πλήθος προκαθορισμένων λειτουργικών διεπαφών. Ειδικά το πακέτο java.util.function αποτελείται αποκλειστικά από λειτουργικές διεπαφές, ενώ υπάρχουν πολλές τέτοιες διεπαφές διασκορπισμένες και στα υπόλοιπα πακέτα. Φυσικά, λειτουργικές διεπαφές μπορούν να οριστούν και από τον χρήστη.
Στο σχήμα 18.1 παρουσιάζονται τέσσερις βασικές προκαθορισμένες λειτουργικές διεπαφές που ανήκουν στο πακέτο java.util.function:



Σχήμα 18.1 *Τέσσερις βασικές λειτουργικές διεπαφές*

Η διεπαφή Predicate<T> δηλώνει την αφηρημένη μέθοδο boolean test(T t). Η Consumer<T> την αφηρημένη μέθοδο void accept(T t). Η Function<T,R> την R apply(T t) και η Supplier<T> την T get().
Στη συνέχεια θα δούμε πώς μπορεί να αξιοποιηθούν αυτές οι διεπαφές σε συνδυασμό με τις εκφράσεις lamda, ξεκινώντας από την Predicate<T>.
Ας υποθέσουμε πως χρειαζόμαστε να τυπώνουμε λίστες από αντικείμενα Person στις οποίες θα περιλαμβάνονται μόνο όσα πρόσωπα πληρούν κάποια ηλικιακά κριτήρια. Πιο συγκεκριμένα, έστω ότι θέλουμε να τυπώσουμε μία λίστα με όσα πρόσωπα είναι ηλικίας άνω των 50 ετών και μία λίστα με όσα πρόσωπα είναι από 40 έως 55 ετών. Μπορούμε να φτιάξουμε μία συνάρτηση που θα παίρνει ως παράμετρο μία λίστα με το σύνολο των προσώπων και ένα αντικείμενο Predicate<Person>. Εφόσον ένα πρόσωπο θα περνά επιτυχώς την test της Predicate θα τυπώνεται. Στον κώδικα 18.1 παρουσιάζεται μία τέτοια συνάρτηση:

```
static void prt(List<Person> list, Predicate<Person> choice) {
    Iterator<Person> it = list.iterator();
    while (it.hasNext()) {
        Person p = it.next();
        if (choice.test(p)) {
            System.out.println(p + " " + p.getAge());
        }
    }
}
```

Κώδικας 18.1 *Εκτύπωση λίστας προσώπων με κριτήρια*

Σύμφωνα με όσα γνωρίζουμε για να καλέσουμε την ptr ώστε να εκπληρώσουμε τον σκοπό μας χρειαζόμαστε δύο κλάσεις όπως δείχνει ο κώδικας 18.2.

```
class SelectorGreater50 implements Predicate<Person> {

    @Override
    public boolean test(Person t) {
        return t.getAge() > 50;
    }
}

class SelectorBetween40and55 implements Predicate<Person> {

    @Override
    public boolean test(Person t) {
       return t.getAge() >=40 && t.getAge()<=55;
    }
```

```
    }
```

Κώδικας 18.2 *Δύο κλάσεις που υλοποιούν τη διεπαφή Predicate*

Έχοντας στη διάθεσή μας τους κώδικες 18.1 και 18.2, μπορούμε να εκπληρώσουμε τον στόχο μας όπως δείχνει ο κώδικας 18.3.

```
public static void main(String[] args) {
    List<Person> list = loadData();
    prt(list, new SelectorGreater50());
    System.out.println("--------------------");
    prt(list, new SelectorBetween40and55());
}
```

Κώδικας 18.3 *Εκτύπωση προσώπων άνω των 50 ετών και εκτύπωση προσώπων από 40 έως και 55 ετών.*

Προσέξτε τώρα τα εξής: Καταρχάς, η διεπαφή Predicate είναι λειτουργική, δηλαδή διαθέτει μία μόνο αφηρημένη μέθοδο, την test. Επομένως, για να ορίσουμε μία κλάση που υλοποιεί τη διεπαφή εκείνο που είναι αναγκαίο είναι ο ορισμός της test. Στη μία περίπτωση ο ορισμός της test δίνεται από τον κώδικα

```
return t.getAge() > 50;
```

και στην άλλη περίπτωση από τον κώδικα

```
return t.getAge() >=40 && t.getAge()<=55;
```

Γιατί όμως να είμαστε αναγκασμένοι να ορίσουμε τις κλάσεις SelectorGreater50 και SelectorBetween40and55, αφού η μόνη πληροφορία που προσθέτουμε σε κάθε κλάση περιορίζεται στον αντίστοιχο κώδικα της test; Εδώ είναι που μας βοηθούν οι εκφράσεις λάμδα κάνοντας τον κώδικά μας συνοπτικό και κομψό, ενώ αυξάνουν την παραγωγικότητά μας. Οι εκφράσεις λάμδα λοιπόν μας επιτρέπουν να περάσουμε στη θέση ενός αντικειμένου λειτουργικής διεπαφής μόνο τον κώδικα που υλοποιεί τη μοναδική αφηρημένη μέθοδο της διεπαφής.
Έτσι, ο κώδικας 18.3 μπορεί να μεταγραφεί με τη χρήση των εκφράσεων λάμδα όπως δείχνει ο κώδικας 18.4.

```
public static void main(String[] args) {
    List<Person> list = loadData();
    prt(list, t->t.getAge()>50);
    System.out.println("----------");
    prt(list, t->t.getAge()>=40 && t.getAge()<=55);
}
```

Κώδικας 18.4 *Χρήση των εκφράσεων λάμδα*

Στην πρώτη κλήση της prt στον κώδικα 18.4, δίνουμε τον κώδικα που υλοποιεί την test για ηλικίες άνω των 50 ετών. Η test γνωρίζουμε πως απαιτεί ως παράμετρο ένα Person, γεγονός που είναι γνωστό και στον μεταγλωττιστή, επομένως δεν είναι αναγκαίο να ορίσουμε τον τύπο της παραμέτρου. Επειδή όμως είναι αναγκαίο να αναφερθούμε σε αυτήν, θα πρέπει να της δώσουμε ένα όνομα. Το όνομα αυτό προηγείται του συμβόλου ->, ενώ ο κώδικας που ακολουθεί το σύμβολο -> είναι ο κώδικας που υλοποιεί την αφηρημένη μέθοδο.
Με τη βοήθεια των εκφράσεων λαμδα είναι εύκολο να τυπώσουμε λίστες προσώπων με οποιαδήποτε κριτήρια ηλικιακά ή μη. Για παράδειγμα, μπορούμε να χρησιμοποιήσουμε την prt για να τυπώσουμε όλα τα πρόσωπα της αρχικής λίστας όπως δείχνει ο κώδικας που ακολουθεί:

```
prt(list, p->true);
```

ή να τυπώσουμε όλα τα πρόσωπα που έχουν επώνυμο Smith

```
prt(list, p->p.getsName().equals("Smith"));
```

Στη συνέχεια δίνουμε παραδείγματα αξιοποίησης των υπόλοιπων τριών λειτουργικών διεπαφών που αναφέρονται σε αυτήν την ενότητα.
Όπως θα έγινε αντιληπτό, η διεπαφή Predicate έχει ως στόχο να λαμβάνει μία παράμετρο, να την αξιολογεί και να επιστρέφει true ή false ανάλογα με κάποια κριτήρια. Αντίθετα, η διεπαφή Consumer έχει ως στόχο να επεξεργάζεται την παράμετρο της αφηρημένης μεθόδου της. Ο κώδικας 18.5 δίνει ένα σχετικό παράδειγμα:

```
import java.util.Iterator;
import java.util.List;
import java.util.function.Consumer;

public class Consumers {

    static void process(List<Person> list, Consumer<Person> consumer) {
        Iterator<Person> it = list.iterator();
        while (it.hasNext()) {
            Person p = it.next();
            consumer.accept(p);
        }
    }

    public static void main(String[] args) {
        List<Person> list = loadData();
        prt(list, p -> true);
        System.out.println("-----------------------");
        process(list, p -> p.setBirthday(p.getBirthday().minusYears(20)));
        prt(list, p -> true);

    }
}
```

Κώδικας 18.5 *Παράδειγμα χρήσης της Consumer*

Η συνάρτηση process του κωδικα 18.5 διασχίζει μια λίστα από αντικείμενα Person και σε κάθε αντικείμενο εφαρμόζει την επεξεργασία της Consumer. Η κλήση της στην main έχει ως αποτέλεσμα τη μεταβολή της ημερομηνίας γέννησης για όλα τα πρόσωπα της λίστας. Πιο συγκεκριμένα, μετά την κλήση της process στην main, όλα τα πρόσωπα της λίστας εμφανίζονται γηραιότερα κατά 20 έτη.
Ακολουθεί παράδειγμα χρήσης της διεπαφής Function:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Function;
import static tmp.FunctionalInterfaces.Predicates.prt;

public class Functions {

    static List<Person> getOlders(List<Person> list,
Function<Person,Boolean> f) {
        List<Person> rVal=new ArrayList<>();
        Iterator<Person> it=list.iterator();
        while (it.hasNext()) {
            Person p=it.next();

            if (f.apply(p)) {
               rVal.add(p);
            }
        }
        return rVal;
    }
```

```
    public static void main(String[] args) {
        List<Person> list = loadData();
        Function<Person, Boolean> ageLimit=p->p.getAge()>50;
        prt(list,p->true);
        System.out.println("-----------------");
        List<Person> olders=getOlders(list, ageLimit);
        prt(olders,p->true);
    }
}
```

Κώδικας 18.6 *Παράδειγμα χρήσης της Function*

Η Function είναι παραμετροποιημένη ως προς δύο τύπους. Η αφηρημένη μέθοδός της, apply, λαμβάνει τον ένα τύπο ως παράμετρο και επιστρέφει τον άλλο. Στον κώδικα 18.6, χρησιμοποιείται Function<Person, Boolean>. Στην main με τον ορισμό της ageLimit, επιδεικνύεται η δυνατότητα ορισμού εκφράσεων λάμδα ως απλές μεταβλητές. Η ageLimit περνά στην getOlders. Η getOlders προσθέτει κάθε Person για το οποίο η apply επιστρέφει true στην λίστα rVal. Τελικά η getOlders επιστρέφει μια λίστα με όλα τα πρόσωπα άνω των 50 ετών.
Στη συνέχεια, στον κώδικα 18.7, με τη βοήθεια της διεπαφής Supplier δημιουργούμε μία μεταβλητή ικανή να παράγει τυχαίους ακέραιους:

```
import java.util.Random;
import java.util.function.Supplier;

public class Suppliers {

    public static void main(String[] args) {
        Supplier<Integer> gen = () -> new Random().nextInt();
        for (int i = 0; i < 10; i++) {
            System.out.println(gen.get());
        }
    }
}
```

Κώδικας 18.7 *Παράδειγμα χρήσης της Supplier*

Προσέξτε πως καθώς η αφηρημένη μέθοδος get δεν δέχεται παραμέτρους, στον ορισμό της έκφρασης λάμδα έχουμε δώσει κενή λίστα παραμέτρων.

18.1.2 Το συντακτικό των εκφράσεων λάμδα

Μία έκφραση λάμδα συντάσσεται ως ακολούθως:

```
λίστα παραμέτρων βέλος κώδικας
```

Η λίστα παραμέτρων μπορεί να είναι κενή ή να περιλαμβάνει μία ή περισσότερες παραμέτρους. Στην περίπτωση που περιλαμβάνει μία μόνο παράμετρο, οι παρενθέσεις μπορούν να παραλειφθούν.

Ο κώδικας μπορεί να είναι μία απλή έκφραση ή ένα μπλοκ κώδικα. Στην περίπτωση που είναι μπλοκ κώδικα και εφόσον η αφηρημένη μέθοδος δεν είναι void, η επιστροφή θα πρέπει να γίνει με χρήση της δεσμευμένης λέξης return.

Οι εκφράσεις λάμδα είναι στην ουσία ένα είδος ανώνυμων συναρτήσεων [2]. Προσέξτε το επόμενο παράδειγμα όπου ταξινομούμε μία λίστα από πρόσωπα πρώτα ως προς το id τους και μετά ως προς επώνυμο και όνομα.

```
import java.util.List;

public class ComparatorLamda {

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

```
        List<Person> list = PolymorphismDemo.loadData();
        list.sort((p1, p2) -> {
            int id1 = Integer.decode(p1.getId());
            int id2 = Integer.decode(p2.getId());

            return id1 - id2;
        });
        prt(list, p -> true);
        System.out.println("--------------------");
        list.sort((p1, p2) -> {
            String sName1 = p1.getsName();
            String sName2 = p2.getsName();
            String fName1 = p1.getfName();
            String fName2 = p2.getfName();
            int rVal = sName1.compareTo(sName2);
            if (rVal == 0) {
                return fName1.compareTo(fName2);
            }
            return rVal;
        });
        prt(list, p -> true);
    }
```

Κώδικας 18.8 *Ταξινόμηση με εκφράσεις λάμδα*

18.1.3 Αναφορές μεθόδων

Εφόσον έχουμε στη διάθεσή μας μία μέθοδο που είναι ταυτόσημη ως προς τη λειτουργικότητα με τον κώδικα μίας έκφρασης λάμδα, τότε η έκφραση λάμδα μπορεί να αντικατασταθεί από την αναφορά της μεθόδου (method reference) [3].

```
import java.util.List;

public class MethodReferences {

    static boolean isOlder(Person p) {
        return p.getAge() > 50;
    }

    static boolean between40And55(Person p) {
        return p.getAge() >= 40 && p.getAge() <= 55;
    }

    public static void main(String[] args) {
        List<Person> list = loadData();
        prt(list, MethodReferences::isOlder);
        System.out.println("----------");
        prt(list, MethodReferences::between40And55);
    }
}
```

Κώδικας 18.9 *Χρήση αναφορών μεθόδων*

Ο κώδικας 18.9 περιλαμβάνει δύο boolean μεθόδους, την isOlder και την between40And55. Με δεδομένες αυτές τις μεθόδους, ο κώδικας 18.4 έχει μετασχηματιστεί στην main του κώδικα 18.9. Με άλλα λόγια, η έκφραση λάμδα

```
t->t.getAge()>50
```

αντικαταστάθηκε από την αναφορά

```
MethodReferences::isOlder
```

και η έκφραση

```
t->t.getAge()>=40 && t.getAge()<=55
```

αντικαταστάθηκε από την αναφορά

```
MethodReferences::between40And55
```

Στο παράδειγμα αυτό οι εκφράσεις λάμδα απαιτούν ως παράμετρο ένα αντικείμενο τύπου Person, την ίδια παράμετρο απαιτούν και οι αναφορές μεθόδων που αντικατέστησαν τις εκφράσεις λάμδα. Επίσης τόσο οι εκφράσεις λάμδα και οι αναφορές μεθόδων επιστρέφουν boolean τιμή. Είναι αυτονόητο πως για να επιτρέψει ο μεταγλωττιστής αντικατάσταση μιας έκφρασης λάμδα, θα πρέπει τόσο οι παράμετροί της όσο και η τιμή επιστροφής της να είναι συμβατές με αυτές της μεθόδου αναφοράς.
Στο παράδειγμα αυτό χρησιμοποιήθηκαν στατικές μέθοδοι. Μπορεί βέβαια να χρησιμοποιηθούν και μέθοδοι στιγμιότυπου. Για παράδειγμα, η πρώτη γραμμή της main στον κώδικα 18.7, δηλαδή ο κώδικας

```
Supplier<Integer> gen = () -> new Random().nextInt();
```

είναι ισοδύναμος με τον κώδικα

```
Supplier<Integer> gen = new Random()::nextInt;
```

Ειδικά, οι αναφορές σε δημιουργούς γίνονται με τη χρήση της δεσμευμένης λέξης new. Για παράδειγμα, ο κώδικας

```
Supplier<String> newString = () -> new String();
```

μπορεί να αντικατασταθεί από τον κώδικα

```
Supplier<String> newString = String::new;
```

Όταν όμως ο δημιουργός διαθέτει παραμέτρους, τότε δεν μπορεί να χρησιμοποιηθεί η διεπαφή Supplier, καθώς η μέθοδος get είναι χωρίς παραμέτρους. Μπορεί όμως να χρησιμοποιηθεί η διεπαφή Function, όπως επιδεικνύει ο κώδικας που ακολουθεί:

```
Function<String, String> s=String::new;
System.out.print(s.apply("Hello"));
System.out.println(s.apply(" Mike"));
```

18.2 Απομνημόνευση

Η απομνημόνευση (memoization) είναι μία τεχνική που εφαρμόζεται σε συνδυασμό με την αναδρομή με σκοπό να περιορίσει το πλήθος των αναδρομικών κλήσεων όπου αυτό είναι δυνατό.
Ας πάρουμε για παράδειγμα τον υπολογισμό του ν-οστού όρου της ακολουθίας Fibonacci. Ο κώδικας 9.4 υπολογίζει αναδρομικά τον ν-οστό όρο. Επίσης, το σχήμα 9.3 μας δίνει μία εικόνα των αναδρομικών κλήσεων. Όπως φαίνεται στο σχήμα 9.3, κατά τον υπολογισμό του όρου με ν=4, η κλήση για n=2 επαναλαμβάνεται δύο φορές. Αν επιχειρήσουμε να υπολογίσουμε τον 20ό όρο της ακολουθίας και μετρήσουμε τις αναδρομικές κλήσεις (Ενότητα 9.7, Άσκηση 5) θα διαπιστώσουμε πως απαιτούνται 21891 αναδρομικές κλήσεις. Αν όμως βρούμε έναν τρόπο ώστε οι τιμές της ακολουθίας να υπολογίζονται μόνο μία φορά η κάθε μία, οι αναδρομικές κλήσεις μειώνονται δραματικά.

```
import java.util.HashMap;
```

```
public class Fibonacci {

    static long cnt1 = 0;
    static long cnt2 = 0;

    static long fibonacciR(int n) {
        cnt1++;
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return fibonacciR(n - 1) + fibonacciR(n - 2);
        }
    }

    static long fibonacciDP(int n, HashMap<Integer, Long> mem) {
        cnt2++;
        if (n == 0 || n == 1) {
            return 1;
        }
        try {
            return mem.get(n);
        } catch (Exception e) {
            long rslt = fibonacciDP(n - 1, mem) + fibonacciDP(n - 2, mem);
            mem.put(n, rslt);
            return rslt;
        }
    }

    public static void main(String[] args) {
        System.out.println("fib(" + 20 + ")=" + fibonacciR(20));
        System.out.println("----------------------");
        System.out.println("fib(" + 20 + ")=" + fibonacciDP(20, new HashMap<>()));
        System.out.println(cnt1 + " " + cnt2);
    }
}
```

Κώδικας 18.10 *Η αναδρομική συνάρτηση Fibonacci χωρίς απομνημόνευση και με απομνημόνευση*

Ο κώδικα 18.10 παρουσιάζει δύο εκδόσεις της Fibonacci. Και οι δύο εκδόσεις είναι αναδρομικές. Η συνάρτηση FibonacciR είναι απλή αναδρομική και η FibonacciDP είναι αναδρομική με απομνημόνευση. Ο μετρητής cnt1 μετράει τις κλήσεις στην απλή αναδρομική και ο cnt2 μετράει τις κλήσεις στην αναδρομική με απομνημόνευση. Η έξοδος του κώδικα είναι:

```
fib(20)=10946
----------------------
fib(20)=10946
21891 39
```

Επομένως προκειμένου να υπολογισθεί ο 20ός όρος της ακολουθίας η απλή αναδρομική συνάρτηση πραγματοποιεί 21891 κλήσεις κι η αναδρομική με απομνημόνευση πραγματοποιεί 39 κλήσεις.

Η FibonacciDP αξιοποιεί ένα HashMap<Integer, Long>. Ο Integer σε αυτό το HashMap αντιπροσωπεύει την τάξη του όρου της ακολουθίας και ο Long την τιμή του αντίστοιχου όρου. Κατά την κλήση της συνάρτησης, γίνεται προσπάθεια να ανακτηθεί από το HashMap η τιμή του ζητούμενου όρου. Εφόσον αυτή δεν βρεθεί, παράγεται εξαίρεση. Κατά τη σύλληψη της εξαίρεσης υπολογίζεται ο όρος και εισάγεται στο HashMap, έτσι την επόμενη φορά που θα ζητηθεί ο ίδιος όρος θα βρεθεί στο HashMap και δεν θα απαιτηθεί ο επανυπολογισμός του.

18.3 Λυμένες ασκήσεις

18.3.1 Εφαρμογή εκφράσεων λάμδα

Δημιουργήστε τρεις εκφράσεις λάμδα. Η πρώτη θα υπολογίζει το άθροισμα των στοιχείων ενός πίνακα πραγματικών, η δεύτερη θα υπολογίζει τον μέσο όρο των στοιχείων ενός πίνακα πραγματικών και η τρίτη το ελάχιστο στοιχείο ενός πίνακα πραγματικών. Εκχωρήστε τις εκφράσεις λάμδα σε μεταβλητές ονομαζόμενες addition, average, minElement, αντίστοιχα. Προσθέστε main στην οποία υπολογίστε το άθροισμα, τον μέσο όρο και το ελάχιστο στοιχείο ενός πίνακα πραγματικών χρησιμοποιώντας τις αντίστοιχες μεταβλητές.

Λύση

```
interface ArrayProcessor {
    double apply( double[] array );
}

public class Processor {

static ArrayProcessor addition= array-> {
    double sum=0;
    for (double d: array)
        sum+=d;
    return sum;
};

static ArrayProcessor average=array-> {
    return addition.apply(array)/array.length;
};

static ArrayProcessor minElement=array-> {
    double min=array[0];
    for (int i=1; i<array.length; i++)
        if (array[i]<min) {
            min=array[i];
        }
    return min;
};

    public static void main(String[] args) {
        double[] array=new double[]{1,2,3,4};
        double sum=addition.apply(array);
        System.out.println(sum);
        System.out.println(average.apply(array));
        System.out.println(minElement.apply(array));
    }

}
```

Κώδικας 18.11 *Υπολογισμοί σε πίνακα πραγματικών με εκφράσεις λάμδα*

Καταρχάς, προκειμένου να ορίσουμε τις εκφράσεις λάμδα, θα πρέπει να έχουμε στη διάθεσή μας κατάλληλη λειτουργική διεπαφή, δηλαδή μία διεπαφή που δηλώνει μία αφηρημένη μέθοδο η οποία λαμβάνει ως παράμετρο έναν πίνακα πραγματικών και επιστρέφει έναν πραγματικό αριθμό. Πράγματι, η διεπαφή ArrayProcessor που δηλώνεται στην αρχή του κώδικα 18.11 πληροί τις απαιτούμενες προδιαγραφές.

Στη συνέχεια, ορίζουμε τις εκφράσεις λάμδα τις οποίες εκχωρούμε στις αντίστοιχες μεταβλητές των οποίων ο τύπος είναι ArrayProcessor. Στην main καλούμε διαμέσου των μεταβλητών την apply δίνοντάς της έναν πίνακα πραγματικών ως παράμετρο.

18.3.2 Αναδρομικός υπολογισμός δύναμης με απομνημόνευση

Υλοποιήστε μια συνάρτηση αναδρομικού υπολογισμού δύναμης με πραγματική βάση υψωμένη σε ακέραιο εκθέτη χωρίς απομνημόνευση και μία με απομνημόνευση. Μελετήστε πώς ακριβώς θα πρέπει να αναπτυχθεί η συνάρτηση με απομνημόνευση ώστε να έχουμε περιορισμό των αναδρομικών κλήσεων.

Λύση

```
class Tupple {

    double b;
    int e;

    public Tupple(double b, int e) {
        this.b = b;
        this.e = e;
    }

    @Override
    public int hashCode() {
        int hash = 3;
        hash = 43 * hash + (int) this.b;
        hash = 43 * hash + this.e;
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Tupple other = (Tupple) obj;
        if (this.b != other.b) {
            return false;
        }
        return (int) this.e == (int) other.e;
    }

}

public class Power {

    static long cnt1 = 0;
    static long cnt2 = 0;
    static HashMap<Tupple, Double> mem = new HashMap<>();

    static double pow(int b, int e) {
        cnt1++;
        if (e > 0) {
            return b * pow(b, e - 1);
        } else if (e < 0) {
            return 1 / pow(b, -e);
        }
        return 1;
    }

    static double fastPow(int b, int e, HashMap<Tupple, Double> mem) {
        cnt2++;
        if (e != 0) {
            try {
                return mem.get(new Tupple(b, e));
```

```
                } catch (Exception x) {
                    double rslt;
                    if (e > 0) {
                        rslt = b * fastPow(b, e - 1, mem);
                    } else {
                        rslt = 1 / fastPow(b, -e, mem);
                    }
                    mem.put(new Tupple(b, e), rslt);
                    return rslt;
                }
            }
            return 1;
        }

        static void test() {
            cnt1=0;
            cnt2=0;
            System.out.println("pow(2,16)=" + pow(2, 16));
            System.out.println("fastpow(2,16)=" + fastPow(2, 16, mem));
            System.out.println(cnt1 + " " + cnt2);
        }

        public static void main(String[] args) {
            test();
            System.out.println("----------");
            test();
        }
    }
```

Κώδικας 18.12 *Υπολογισμός δύναμης χωρίς και με απομνημόνευση*

Η έξοδος του κώδικα 18.2 έχει ως εξής;

```
pow(2,16)=65536.0
fastpow(2,16)=65536.0
17 17
----------
pow(2,16)=65536.0
fastpow(2,16)=65536.0
17 1
```

Αυτό σημαίνει πως κατά την πρώτη κλήση της test και οι δύο εκδόσεις του υπολογισμού της δύναμης πραγματοποίησαν από 17 αναδρομικές κλήσεις. Ωστόσο, κατά τη δεύτερη κλήση της test, η έκδοση χωρίς απομνημόνευση πραγματοποίησε και πάλι 17 αναδρομικές κλήσεις, ενώ η έκδοση με απομνημόνευση μόνο την αρχική κλήση στην fastPow. Πράγματι κατά τον αναδρομικό υπολογισμό της δύναμης δεν επαναλαμβάνεται κλήση με ίδιες παραμέτρους. Αν για παράδειγμα θελήσουμε να υπολογίσουμε το 2^3, θα πρέπει να υπολογιστούν οι δυνάμεις 2^2, 2^1 και 2^0. Επομένως, κανένας υπολογισμός δεν επαναλαμβάνεται. Ωστόσο, αν έχουμε μία συνάρτηση υπολογισμού δύναμης, είναι σχεδόν βέβαιο ότι θα την χρησιμοποιήσουμε επαναληπτικά σε μία εφαρμογή. Τότε προκύπτει το όφελος. Όμως αυτό το όφελος δεν θα προκύψει αν κατά την κλήση της συνάρτησης περάσουμε μία νέα απεικόνιση ως παράμετρο, όπως κάναμε με τη συνάρτηση Fibonacci. Αντίθετα, χρειαζόμαστε μια στατική απεικόνιση στην ίδια κλάση που περιλαμβάνει την αναδρομική συνάρτηση ώστε τα αποτελέσματα των υπολογισμών να είναι δεδομένα μεταξύ διαφορετικών κλήσεων.

Βιβλιογραφία

[1] S. G. Ganesh, H. K. Kumar, and T. Sharma, Oracle Certified Professional Java SE 8 Programmer Exam 1Z0-809: A Comprehensive OCPJP 8 Certification Guide, 2nd ed. USA: Apress, 2015.

[2] “Lambda Expressions (The Java™ Tutorials > Learning the Java Language > Classes and Objects).” https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax (accessed Jan. 04, 2022).

[3] “Method References (The Java™ Tutorials > Learning the Java Language > Classes and Objects).” https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html (accessed Jan. 05, 2022).

Κριτήρια Αξιολόγησης

Αμεταβλητότητα

Έστω ο κώδικας

```
public class KA01 {

	public static void main(String args[]) {
		String t = "12345";
		t.replace("2", "A");
		System.out.println(t);
	}
}
```

Ποια είναι η έξοδός του;

```
A. 12345
B. 1A345
Γ. 12A45
```

Απάντηση/Λύση

Όπως έχουμε εξηγήσει, τα Strings είναι αμετάβλητα. Από τη στιγμή που θα δημιουργηθούν, διατηρούν την τιμή τους μέχρι το πέρας της εφαρμογής. Οι συναρτήσεις της κλάσης String, όπως η replace δεν μεταβάλλουν το String από το οποίο καλούνται, αλλά δημιουργούν και επιστρέφουν ένα νέο String. Το String που επέστρεψε η replace δεν το αξιοποιεί ο κώδικας της ερώτησης. Η δε αναφορά s συνεχίζει να δείχνει στο String “12345”. Έτσι, η σωστή απάντηση είναι η Α.

Αρχικοποίηση

Έστω ο κώδικας

```
public class KA02 {

	static double x;

	public static double calc(double value) {
		int a, b, c;
		if (x < 10) {
			a = 2;
			b = 3;
			c = 4;
		}
		return a * b * c * value;
	}

	public static void main(String[] args) {
		System.out.println(calc(100));
	}
}
```

Ποια είναι η έξοδός του;

```
A. 240
B. 2400
Γ. Λάθος μεταγλώττισης
Δ. 0
```

Απάντηση/Λύση

Η στατική μεταβλητή x ως μέλος της κλάσης αρχικοποιείται αυτόματα στο 0. Επομένως, αν εκτελεστεί η if μέσα στην calc, η έκφραση x<10 θα είναι αληθής και άρα ο κώδικας θα εισέλθει μέσα στο μπλοκ της if και θα αρχικοποιήσει τις a, b, c. Ωστόσο, αυτό είναι κάτι που δεν μπορεί να υπολογιστεί κατά τον χρόνο μεταγλώττισης, αλλά μόνο κατά τον χρόνο εκτέλεσης. Παράλληλα, οι τοπικές μεταβλητές σε αντίθεση με τα μέλη της κλάσης δεν αρχικοποιούνται αυτόματα. Επομένως, για τον μεταγλωττιστή, στην return της calc, οι μεταβλητές a, b, c ενδέχεται να είναι αναρχικοποίητες, αποκρίνεται με λάθος μεταγλώττισης και η σωστή απάντηση είναι η Γ.

Η main

Τι θα συμβεί αν επιχειρήσουμε να μεταγλωττίσουμε τον ακόλουθο κωδικα;

```
public class KA03 {

     public static int main(String[] args) {
          System.out.println("Hello World!");
          return 0;
     }
}
```

```
A. Λάθος μεταγλώττισης
B. Θα μεταγλωττιστεί και θα τυπώσει Hello World!
Γ. Θα μεταγλωττιστεί και θα τυπώσει 0
Δ. Θα μεταγλωττιστεί αλλά δεν θα τυπώσει τίποτα.
```

Απάντηση/Λύση

Ο κώδικας δεν έχει κάποιο θέμα με τη μεταγλώττιση, οπότε η επιλογή Α δεν ισχύει. Επομένως, ο κώδικας θα μεταγλωττιστεί. Κατά την εκτέλεση θα αναζητηθεί η main. Ο κώδικας περιλαμβάνει μία main, ωστόσο είναι διαφορετική από την main που λειτουργεί ως κύρια είσοδος σε εφαρμογή Java. Η μεν main του κώδικα επιστρέφει int, η δε main που αναγνωρίζεται ως κύρια είσοδος στο πρόγραμμα είναι τύπου void. Επομένως, ο κώδικας δεν θα τρέξει, άρα αποκλείονται η Β και η Γ και η σωστή απάντηση είναι η Δ. Αν επιχειρήσουμε να τρέξουμε τον κώδικα, η JVM θα αποκριθεί με μήνυμα πως δεν βρήκε main.

Μεταβλητές και Συναρτήσεις στιγμιότυπου

Ο κώδικας

```
class Alpha {

     public int h = 1;

     public int getH() {
          return h;
     }
}

public class KA04 extends Alpha {
```

```
        public int h = 11;

        @Override
        public int getH() {
              return h;
        }

        public static void main(String[] args) {
              Alpha parent = new KA04();
              System.out.print(parent.h + "-" + parent.getH() +
              "->");
              KA04 child = (KA04) parent;
              System.out.println(child.h + "-" + child.getH());
        }
  }
```

θα εμφανίσει

```
A. 1-11->11-11
B. 11-11->11-11
Γ. 1-11->1-11
Δ. Τίποτε από τα παραπάνω
```

Απάντηση/Λύση

Εδώ έχουμε δύο αναφορές, την parent και την child. Ωστόσο και οι δύο αναφορές δείχνουν στο ίδιο πραγματικό αντικείμενο, αυτό που δημιουργείται στην πρώτη γραμμή της main. Άσχετα από τον τύπο της αναφοράς, η συνάρτηση getH που θα τρέξει είναι η συνάρτηση του πραγματικού και όχι του τυπικού αντικειμένου. Επομένως, η συνάρτηση θα εμφανίσει και τις 2 φορές που καλείται την τιμή 11. Δεν ισχύει το ίδιο όμως με τη μεταβλητή. Η μεταβλητή θα επιλυθεί με βάση τον τύπο του αντικειμένου και όχι με βάση το πραγματικό αντικείμενο. Έτσι η σωστή απάντηση είναι η Α. Αν ωστόσο στην κλάση KA04 δεν είχε οριστεί μεταβλητή h, τότε σωστή θα ήταν η απάντηση Γ.

Αξιολόγηση βραχέως κυκλώματος

Τι θα εμφανίσει ο ακόλουθος κώδικας κατά την εκτέλεσή του;

```
class KA05 {

      static boolean a;
      static boolean b;
      static boolean c;

      public static void main(String[] args) {
            boolean bool = (a = true) || (b = true) && (c =
            true);
            System.out.println(a + ", " + b + ", " + c);
      }
}
```

```
A. true, true, true
B. true, false, false
Γ. true, false, true
Δ. false, false, false
Ε. Λάθος μεταγλώττισης
```

Απάντηση/Λύση

Τα στατικά μέλη a, b, c αρχικοποιούνται αυτόματα στην τιμή false. Γενικότερα, ο κώδικας δεν περιέχει κάποιο λάθος μεταγλώττισης, οπότε η απάντηση Ε είναι λάθος. Στην main η παράσταση που εκχωρείται στην bool αξιολογείται από αριστερά προς τα δεξιά. Προσέξτε πως η παράσταση αυτή είναι μία σειρά από εκχωρήσεις και όχι λογικές συγκρίσεις που λειτουργούν ως τελεστές λογικών πράξεων. Στη μεταβλητή a επομένως εκχωρείται η τιμή true και το αποτέλεσμα της (a = true) είναι true. Εφόσον η τιμή true συνδέεται με τον τελεστή ||, γνωρίζουμε πως λόγω της αξιολόγησης βραχέως κυκλώματος, το υπόλοιπο τμήμα της παράστασης δεν θα αξιολογηθεί. Συνεπώς, στις μεταβλητές b και c δεν εκχωρείται άλλη τιμή. Αυτές διατηρούν την αρχική τιμή τους, δηλαδή την τιμή false. Επομένως, σωστή απάντηση είναι η Β.

Μεταβλητές τιμής

Ο κώδικας

```
public class KA06 {

     int k = 5;

     public boolean checkIt(int k) {
          return k-- > 0;
     }

     public void printThem() {
          while (checkIt(k)) {
               System.out.print(k);
          }
     }

     public static void main(String[] args) {
          new KA06().printThem();
     }
}
```

εμφανίζει

```
Α. 54321
Β. 43210
Γ. Λάθος μεταγλώττισης
Δ. Λάθος χρόνου εκτέλεσης
```

Απάντηση/Λύση

Η printThem καλείται στην main. Αυτή σε μια επαναληπτική διαδικασία που ελέγχεται από την checkIt(k) τυπώνει την τιμή του k. Μέσα στην checkIt μειώνεται η τιμή του k κατά 1. Ωστόσο, το k μέσα στην checkIt δεν είναι το ίδιο με το k μεταβλητή στιγμιότυπου της κλάσης. Το k μέσα στην checkIt είναι παράμετρος τιμής και λειτουργεί ως τοπική μεταβλητή αφήνοντας αμετάβλητη την τιμή της μεταβλητής του στιγμιότυπου. Επομένως, η checkIt με κάθε κλήση της τυπώνει το 5 και το πρόγραμμα πέφτει σε ατέρμονη επαναληπτική διαδικασία, άρα η σωστή απάντηση είναι η Δ.

Δισδιάστατοι πίνακες

Ποια είναι η έξοδος της main;

```
public class KA07 {

     public static void main(String[] args) {
          int[][] t = {{1, 2}, {3, 4}, {5, 6}};
```

```
            for (int i = 0; i < t.length; i++) {
                for (int j = 0; j < t.length; j++) {
                    System.out.print(t[i][j]);
                }
            }
        }
}
```

```
A. 12345
B. Λάθος μεταγλώττισης
Γ. Λάθος χρόνου εκτέλεσης
```

Απάντηση/Λύση

Όπως έχουμε εξηγήσει, οι δισδιάστατοι πίνακες είναι στην ουσία πίνακες που το κάθε στοιχείο τους είναι ένας μονοδιάστατος πίνακας. Ο πίνακας t περιέχει τους πίνακες {1, 2}, {3, 4} και {5, 6}. Οι πίνακες αυτοί έχουν μήκος δύο. Από την άλλη πλευρά, ο ίδιος ο πίνακας t έχει μήκος 3. Στον εμφωλευμένο βρόχο όμως χρησιμοποιούμε το μήκος του t και στις δύο for. Επομένως, ο κώδικας θα επιχειρήσει να προσπελάσει καταρχάς το στοιχείο t[0][2], δηλαδή το στοιχείο στη θέση 2 του πίνακα {1, 2}. Τέτοιο στοιχείο όμως δεν υπάρχει, καθώς η τελευταία θέση του πίνακα είναι η 1. Το πρόγραμμα θα πέσει με ArrayIndexOutOfBoundsException και η σωστή απάντηση είναι η Γ.

Ο προκαθορισμένος Δημιουργός

Ποια είναι η έξοδος της main;

```
class A {
    int i;
    public void A() {i=1;}
}

public class KA08 {
    public static void main(String[] args) {
        A a1=new A();
        System.out.println(a1.i);
    }
}
```

```
A. 0
B. 1
Γ. Λάθος μεταγλώττισης
Δ. Λάθος εκτέλεσης
```

Απάντηση/Λύση

Καταρχάς δεν υπάρχει λάθος μεταγλώττισης ούτε και εκτέλεσης. Στην main καλείται ο προκαθορισμένος δημιουργός της κλάσης Α και δημιουργείται το στιγμιότυπο a1. Προσέξτε όμως πως η Α() που ορίζεται στην κλάση Α δεν είναι δημιουργός, καθώς έχει χαρακτηριστεί ως void αλλά μία συνάρτηση στιγμιότυπου. Επομένως κατά τη δημιουργία του a1 δεν τρέχει η συνάρτηση Α(), αλλά ο προκαθορισμένος δημιουργός. Αυτός βέβαια δεν μεταβάλλει την τιμή της i η οποία έχει αρχικοποιηθεί αυτόματα στην τιμή 0. Έτσι ο κώδικας εμφανίζει 0 και σωστή απάντηση είναι η Α.

Ψηφιολέξεις

Ποια η έξοδος της main;

```
public class KA09 {
```

```
    public static void main(String[] args) {
        int i=Integer.MAX_VALUE;
        int j=i+1;
        int k=~i;
        System.out.print(j==k);
        System.out.println(j==Integer.MIN_VALUE);
    }
}

A. truefalse
B. falsetrue
Γ. truetrue
Δ. falsefalse
```

Απάντηση/Λύση

Ο ακέραιος στην Java είναι 32 bit. Για ευκολία όμως θα υποθέσουμε πως ο ακέραιος στην Java είναι 8 bit. Μια τέτοια υπόθεση δεν επηρεάζει καθόλου το σκεπτικό της απάντησης.

Ποια είναι λοιπόν η αναπαράσταση της μέγιστης τιμής ενός προσημασμένου ακέραιου μήκους 8 bits; Καταρχάς, εφόσον πρόκειται για προσημασμένο ακέραιο, γνωρίζουμε ότι το πρώτο bit αναπαριστά το πρόσημο. Εφόσον μας ενδιαφέρει η μέγιστη τιμή, ο αριθμός είναι θετικός και το πρώτο bit είναι 0. Για να πάρουμε λοιπόν τη μέγιστη τιμή, όλα τα υπόλοιπα bits θα πρέπει να λάβουν τη μέγιστη δυνατή τιμή, δηλαδή την τιμή 1. Άρα ο μέγιστος ακέραιος στα 8 bits είναι ο 01111111. Σημειώστε πως στην πράξη ο ψηφιοχάρτης της μεταβλητής i είναι 0 ακολουθούμενο από 31 άσους.

Η σταθερά 1 έχει προφανώς ως ψηφιοχάρτη την τιμή 00000001. Επομένως, στο j εκχωρείται το άθροισμα 01111111+00000001 που μας δίνει 10000000. Εφόσον το πρώτο bit του αθροίσματος είναι 1 πρόκειται για αρνητικό αριθμό. Εφόσον το μέτρο του είναι 0000000 πρόκειται προφανώς για τον μικρότερο αριθμό που είναι δυνατόν να αναπαρασταθεί με 7 bits, δηλαδή στην i έχει εκχωρηθεί ο τιμή Integer.MIN_VALUE που είναι ίση με 10000000. Στην k εκχωρείται η ίδια τιμή ως συμπλήρωμα του i. Επομένως, η έξοδος του κώδικα είναι truetrue και σωστή είναι η απάντηση Γ.

Πράξεις με πραγματικούς

Ποια η έξοδος της main;

```
public class KA10 {

    public static void main(String[] args) {
        System.out.println((Double.MIN_VALUE < 0) + " " +
(Double.NEGATIVE_INFINITY + Double.POSITIVE_INFINITY) + " " +
(258. / 0));
    }
}

A. true 0 Infinity
B. true NaN Infinity
Γ. false 0 Infinity
Δ. false NaN Infinity
Ε. Λάθος χρόνου εκτέλεσης, διαίρεση με το 0
```

Απάντηση/Λύση

Η διαίρεση με το 0 στην έκφραση 258. / 0 δεν παράγει λάθος χρόνου εκτέλεσης. Καθώς ο ένας από τους δύο τελεστέους είναι τύπου double (ο 258.), η διαίρεση είναι διαίρεση πραγματικών. Μάλιστα καθώς το 258. είναι θετικός το αποτέλεσμα είναι +Infinity. Ο τελεστής + όμως παραλείπεται και έτσι η έκφραση 258./0 αξιολογείται σε infinity. Η Double.MIN_VALUE εκφράζει την ελάχιστη θετική τιμή που μπορεί να αναπαρασταθεί από τον τύπο Double. Επομένως η έκφραση Double.MIN_VALUE < 0 επιστρέφει false.

Τέλος, η πρόσθεση Double.NEGATIVE_INFINITY + Double.POSITIVE_INFINITY επιστρέφει NaN και η σωστή απάντηση είναι η Δ.

Σύλληψη εξαιρέσεων

Ποια η έξοδος της main;

```
class MyException extends Exception {
}

class MyException1 extends MyException {
}

public class KA11 {

    void myMethod() throws MyException {
        throw new MyException();
    }


    public static void main(String[] args) throws Exception {
        KA11 et = new KA11();
        try {
            et.myMethod();
        } catch (MyException me) {
            System.out.print("MyException");
        } catch (MyException1 me) {
            System.out.print ("MyException1");
        } finally {
            System.out.print (" finally");
        }
    }
}

A. MyException
B. MyException1
Γ. MyExceptionfinally
Δ. MyException1finally
Ε. Λάθος χρόνου μεταγλώττισης
```

Απάντηση/Λύση

Στο catch συλλαμβάνουμε πρώτα εξαίρεση τύπου MyException και μετά εξαίρεση τύπου MyException1. Ωστόσο, η MyException1 είναι παράγωγη της MyException. Με άλλα λόγια η MyException1 είναι MyException. Έτσι είτε παραχθεί MyException είτε παραχθεί MyException1 θα συλληφθεί από το πρώτο catch. Αυτό σημαίνει ότι ο κώδικας στο δεύτερο catch δεν εκτελείται ποτέ, πράγμα που εντοπίζει ο μεταγλωττιστής και παράγει κατάλληλο μήνυμα. Η σωστή απάντηση είναι η Ε.

Ο κώδικας μπορεί να περάσει με επιτυχία μεταγλώττιση αν αφαιρεθεί το δεύτερο catch. Σε αυτήν την περίπτωση θα έχουμε ενιαία αντιμετώπιση των δύο τύπων εξαιρέσεων. Αν ωστόσο χρειαζόμαστε να διαφοροποιήσουμε μεταξύ MyException και MyException1, τότε θα πρέπει να ελέγξουμε πρώτα την MyException1 και στη συνέχεια την MyException.

instanceof

Ποια η έξοδος της main;

```
class A1{};
class B1 extends A1 {}
class C1 extends B1 {}

public class KA12 {
    public static void main(String[] args) {
        A1 o = new C1();
        System.out.print(o instanceof A1);
        System.out.print(o instanceof B1);
        System.out.print(o instanceof C1);
    }
}

A. truefalsetrue
B. falsefalsetrue
Γ. truefalsefalse
Δ. truetruetrue
```

Απάντηση/Λύση

Ο τελεστής instanceof ελέγχει το πραγματικό αντικείμενο. Στην αναφορά έχει εκχωρηθεί αντικείμενο τύπου C1. Ένα τέτοιο αντικείμενο είναι ένα C1, επίσης είναι ένα B1 και είναι ένα A1. Επομένως, ο τελεστής θα επιστρέψει true και στους τρεις ελέγχους και η σωστή απάντηση είναι η Δ.

Υπερφόρτωση συναρτήσεων

Ποια η έξοδος του κώδικα που ακολουθεί;

```
public class KA13 {

    static void m(int a) { System.out.print("int"); }
    static void m(char a) {System.out.print("char"); }

    public static void main(String[] args) {
        int i = 'a';
        char k = 127;
        m(i);
        m(k);
    }
}

A. Λάθος μεταγλώττισης
B. Λάθος εκτέλεσης
Γ. intchar
Δ. charint
```

Απάντηση/Λύση

Καταρχάς δεν υπάρχει ούτε λάθος μεταγλώττισης ούτε εκτέλεσης. Στη μεταβλητή i εκχωρείται η σταθερά ‘a’. Η σταθερά είναι τύπου χαρακτήρα. Οι χαρακτήρες όπως έχουμε εξηγήσει είναι ακέραιοι τύποι. Εκείνο που αποθηκεύεται είναι ο ακέραιος κωδικός του χαρακτήρα. Κατά την εκχώρηση i = ‘a’ λοιπόν στο i εκχωρείται μία ακέραια τιμή. Άσχετα από το ότι η τιμή αυτή προήλθε από έναν χαρακτήρα όταν προσπελαύνεται μέσω μιας ακέραιας μεταβλητής ερμηνεύεται σαν ακέραιος. Αντίστοιχα λειτουργεί και η εκχώρηση k = 127. Άσχετα με το ότι στην k εκχωρήσαμε μια ακέραιη τιμή, η k παραμένει τύπου χαρακτήρα. Επομένως, η επίλυση των κλήσεων της m στην main είναι σαφής. Στην κλήση m(i) καλείται η m με παράμετρο int και

στην κλήση m(k) καλείται η m με παράμετρο char. Ο κώδικας εμφανίζει intchar και σωστή είναι η απάντηση Γ.

Σειρά αξιολόγησης

Ποια η έξοδος του κώδικα που ακολουθεί;

```
public class KA14 {
    public static void main(String[] args) {
        String s="John";
        String s1=new String(s);
        System.out.print((s==s1)+" "+s.equals(s1));
        System.out.print(" ");
        System.out.print(s==s1+" "+s.equals(s1));
    }
}

A. false true false true
B. false true true false
Γ. false false false
Δ. false true false
```

Απάντηση/Λύση

Η πρώτη print εμφανίζει false true. Η έκφραση (s==s1) είναι false, καθώς με τον τελεστή ισότητας συγκρίνονται οι αναφορές s και s1 που είναι διαφορετικές. Στη συνέχεια, η έκφραση s.equals(s1) είναι true καθώς s και s1 δείχνουν σε διαφορετικά μεν Strings που έχουν όμως και τα δύο το “John” ως περιεχόμενο.

Στη δεύτερη print όμως τα πράγματα είναι διαφορετικά. Εδώ η έκφραση s==s1 δεν περικλείεται σε παρενθέσεις, άρα δεν έχει προτεραιότητα. Ο τελεστής + έχει υψηλότερη προτεραιότητα από τον τελεστή ==. Επομένως, για να υπολογιστεί η έκφραση s==s1+” “+s.equals(s1) θα πρέπει πρώτα να υπολογιστεί η έκφραση s1+” “+s.equals(s1). Η έκφραση αυτή είναι τύπου String και η τιμή της είναι John true. Στη συνέχεια συγκρίνεται η αναφορά s με την ανώνυμη αναφορά στο String John true, οπότε η print εμφανίζει false. Συνεπώς σωστή είναι η απάντηση Γ.

Παραγωγή εξαίρεσης στο finally μπλοκ

Τι μήνυμα θα λάβουμε στη στοίβα κλήσης κατά την εκτέλεση του κώδικα που ακολουθεί;

```
class RTException extends RuntimeException {
    RTException(String msg) {
        super();
    }
}
public class KA15 {

    static void f() throws Exception {
        try {
            throw new RTException("Inside try");
        } catch (RuntimeException e) {
            throw new Exception("Inside catch");
        } finally {
            throw new RuntimeException("Inside finally");
        }
    }

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

```
            f();
        }
    }
```

```
A. Inside try
B. Inside catch
Γ. Inside finally
```

Απάντηση/Λύση

Η παραγωγή εξαίρεσης στο finally κρύβει την κύρια εξαίρεση. Επομένως η εξαίρεση που θα φανεί κατά την εκτύπωση της στοίβας κλήσης είναι αυτή που παρήχθη μέσα στο finally, η οποία τυπώνει Inside finally. Σωστή είναι η απάντηση Γ. Το ζήτημα αυτό αντιμετωπίζεται με τη χρήση της try with resources.

Συλλογή απορριμμάτων

Έστω ο κώδικας

```
 1 package gr.ihu.cs.lmous.OOJ.kritirio2;
 2
 3 public class KA16 {
 4     private String s;
 5
 6     public KA16(String s) {
 7         this.s = s;
 8     }
 9
10     public static void f() {
11         KA16 k1=new KA16("1");
12         KA16 k2=new KA16("2");
13         k1=k2;
14         k2=null;
15         System.out.println("Hello");
16     }
17
18     public static void main(String[] args) {
19         f();
20     }
21 }
```

Επιλέξτε όλες τις σωστές προτάσεις

```
Α. Το στιγμιότυπο KA16 που δημιουργήθηκε στη γραμμή 11 είναι
διαθέσιμο προς συλλογή από τον συλλέκτη απορριμμάτων στη
γραμμή 14
Β. Το στιγμιότυπο KA16 που δημιουργήθηκε στη γραμμή 12 είναι
διαθέσιμο προς συλλογή από τον συλλέκτη απορριμμάτων στη
γραμμή 15
Γ. Με το πέρας της εκτέλεσης της f είναι διαθέσιμο προς
συλλογή από τον συλλέκτη απορριμμάτων μόνο το στιγμιότυπο
KA16 που δημιουργήθηκε στη γραμμή 11.
Δ. Με το πέρας της εκτέλεσης της f είναι διαθέσιμα προς
συλλογή από τον συλλέκτη απορριμμάτων και τα δύο στιγμιότυπα
KA16 που δημιουργήθηκαν στην f.
```

Απάντηση/Λύση

Το στιγμιότυπο ΚΑ16 που δημιουργήθηκε στη γραμμή 11 έχει ως μοναδική αναφορά τη μεταβλητή k1. Η μεταβλητή k1 όμως στη γραμμή 13 παύει να αναφέρεται σε αυτό το στιγμιότυπο που μένει πλέον χωρίς αναφορά. Επομένως, στη γραμμή 14 είναι διαθέσιμο προς συλλογή και άρα η Α είναι σωστή.

Στη γραμμή 14 η αναφορά k2 παύει να δείχνει στο στιγμιότυπο που δημιουργήθηκε στη γραμμή 12. Ωστόσο, στη γραμμή 13, η αναφορά k1 είχε ανακατευθυνθεί προς αυτό το στιγμιότυπο. Πράγματι η αναφορά k2 που έδειχνε αρχικά σε αυτό το στιγμιότυπο, μετά τη γραμμή 14 παύει να το δείχνει. Ωστόσο, μπορούμε να αναφερθούμε σε αυτό μέσω της αναφοράς k1. Επομένως, το στιγμιότυπο δεν είναι διαθέσιμο για να συλλεγεί ως απόρριμμα και η πρόταση Β είναι λανθασμένη.

Με το πέρας της εκτέλεσης της f, οι τοπικές μεταβλητές k1 και k2 καταστρέφονται, επομένως και τα δύο στιγμιότυπα που δημιουργήθηκαν μέσα στην f μένουν χωρίς αναφορά και άρα είναι διαθέσιμα προς συλλογή από τον συλλέκτη απορριμμάτων. Συνεπώς, η πρόταση Γ είναι λανθασμένη και η πρόταση Δ είναι σωστή.

Λίστα παραμέτρων μεταβλητού μήκους

Ποια η έξοδος του κώδικα που ακολουθεί;

```
import java.util.Arrays;

public class KA17 {

    void f(int[] a) {
        System.out.print(a[0]);
    }

    void f(int... a) {
        System.out.print(a[1]);
    }

    public static void main(String[] args) {
        KA17 k = new KA17();
        int[] t = {0, 1, 2, 3};
        k.f(t);
        k.f(0,1,2,3);
    }
}
```

```
A. 01
B. 10
Γ. Λάθος μεταγλώττισης
Δ. Λάθος χρόνου εκτέλεσης
```

Απάντηση/Λύση

Ο μεταγλωττιστής δεν μπορεί να διαφοροποιήσει μεταξύ των δύο συναρτήσεων f, καθώς είναι δυνατό και οι δύο να κληθούν με πραγματική παράμετρο ένα πίνακα ακέραιων. Σωστή απάντηση είναι η Γ.

Auto boxing

Ποια η έξοδος του κώδικα που ακολουθεί;

```
public class KA18 {
    public void f(String s) {
        System.out.print("String");
    }
    public void f(Object s) {
```

```
        System.out.print("Object");
    }

    public static void main(String[] args) {
        KA18 k=new KA18();
        k.f("Hello");
        k.f(1);
    }
}

A. StringObject
B. ObjectString
Γ. Λάθος μεταγλώττισης
Δ. Λάθος χρόνου εκτέλεσης
```

Απάντηση/Λύση

Δεν υπάρχει λάθος μεταγλώττισης, ούτε εκτέλεσης. Στην πρώτη κλήση είναι σαφές ότι ο μεταγλωττιστής επιλέγει την f με παράμετρο String, οπότε εμφανίζεται η σειρά String. Στη δεύτερη κλήση, η f καλείται με παράμετρο 1, δηλαδή με παράμετρο τύπου int. Εφόσον δεν υπάρχει f με παράμετρο int, ο μεταγλωττιστής μετατρέπει το 1 σε Integer. Η διαδικασία αυτή είναι γνωστή ως auto boxing. Είναι η ίδια λειτουργία που μας επιτρέπει να γράφουμε προτάσεις όπως:

```
Integer i=1;
```

Βεβαίως δεν υπάρχει f με παράμετρο Integer. Υπάρχει όμως f με παράμετρο Object. Καθώς ένας Integer είναι ένα Object, καλείται η f με παράμετρο Object. Επομένως, σωστή είναι η απάντηση Α.

Υπερφόρτωση και θεμελιώδεις τύποι

Ποια η έξοδος του κώδικα που ακολουθεί;

```
public class KA19 {
    public void f(int i) {
        System.out.print("int");
    }

    public void f(long l) {
        System.out.print("long");
    }

    public static void main(String[] args) {
        KA19 k=new KA19();
        k.f(1);
        int j=Integer.MAX_VALUE;
        k.f(j+1);
    }
}

A. intlong
B. intint
Γ. Λάθος μεταγλώττισης
```

Απάντηση/Λύση

Δεν υπάρχει λάθος μεταγλώττισης. Η πρώτη κλήση της f γίνεται σαφώς με ακέραια παράμετρο, οπότε ο μεταγλωττιστής καλεί την f(int) και εμφανίζει int. Κατά τη δεύτερη κλήση προστίθεται 1 στη μέγιστη ακέραια τιμή. Σε αυτήν την περίπτωση έχουμε υπερχείλιση, ωστόσο ο τύπος του αποτελέσματος είναι και

πάλι int, οπότε καλείται και πάλι η f(int) και εμφανίζεται int. Επομένως, συνολικά εμφανίζεται intint και σωστή είναι η απάντηση Β.

Αλυσιδωτή κλήση

Ποια η έξοδος του κώδικα που ακολουθεί;

```
public class KA20 {

    int i=0;

    public final KA20 prt() {
        System.out.print(i++);
        return this;
    }

    public final KA20 prt(int j) {
        System.out.print(i);
        return new KA20();
    }

    public static void main(String[] args) {
        new KA20().prt().prt(1).prt(2);
    }
}

A. 1
B. 000
Γ. 010
Δ. 011
E. 012
```

Απάντηση/Λύση

Στην main δημιουργείται ένα ανώνυμο στιγμιότυπο της κλάσης KA20. Από αυτό καλείται η prt χωρίς παραμέτρους η οποία εμφανίζει το i που έχει αρχικοποιηθεί στο 0. Μετά την εμφάνισή του, το i είναι αυξημένο κατά 1 καθώς χρησιμοποιείται ο μεταθεματικός τελεστής προσαύξησης. Η κλήση στην prt() τερματίζεται με return this, δηλαδή η prt() επιστρέφει το στιγμιότυπο από το οποίο κλήθηκε και στο οποίο η μεταβλητή i εχει την τιμή 1. Με άλλα λόγια, η έκφραση new KA20().prt() έχει εμφανίσει 0, έχει μεταβάλλει την τιμή του i σε 1 και έχει επιστρέψει το στιγμιότυπο από το οποίο κλήθηκε. Από αυτό το στιγμιότυπο καλείται η prt με παράμετρο int. Αυτή καταρχάς εμφανίζει το i, δηλαδή 1 και επιστρέφει ένα νέο ανώνυμο αντικείμενο KA20. Επομένως, η έκφραση new KA20().prt().prt(1) επιστρέφει ένα νέο στιγμιότυπο. Η μεταβλητή i στο νέο στιγμιότυπο αρχικοποιείται στο 0. Από αυτό το νέο στιγμιότυπο καλείται και πάλι η prt(int) και εμφανίζει 1. Σημειώστε πως η prt(int) δεν χρησιμοποιεί την παράμετρό της. Αυτή χρησιμοποιείται μόνο από τον μεταγλωττιστή για να διαφοροποιήσει μεταξύ των δύο prt. Η σωστή απάντηση είναι η Γ.

Στατικές μέθοδοι

Επιλέξτε όσες προτάσεις είναι ορθές σε σχέση με τον κώδικα που ακολουθεί:

```
public class KA21 {

    public static void f() {
        System.out.print("f");
    }
```

```
    public void g() {
        System.out.print("g");
    }

    public static void k() {
        f();
        g();
    }

    public static void main(String[] args) {
        KA21 kA = new KA21();
        kA.k();
        KA21 kB=null;
        kB.k();
    }
}
```

```
Α. Ο κώδικας μεταγλωττίζεται όπως έχει.
Β. Υπάρχει ένα μόνο λάθος μεταγλώττισης στον κώδικα.
Γ. Υπάρχουν ακριβώς δύο λάθη μεταγλώττισης στον κώδικα.
Δ. Αν οι γραμμές που περιέχουν λάθη μεταγλώττισης διαγραφούν,
η έξοδος του κώδικα θα είναι ff.
Ε. Αν οι γραμμές που περιέχουν λάθη μεταγλώττισης διαγραφούν,
η έξοδος του κώδικα θα είναι gg.
ΣΤ. Αν οι γραμμές που περιέχουν λάθη μεταγλώττισης διαγραφούν,
ο κώδικας θα παράξει NullPointerException.
```

Απάντηση/Λύση

Η στατική μέθοδος k καλεί τη μη στατική μέθοδο g. Αυτό δεν επιτρέπεται, καθώς η k μπορεί να κληθεί και χωρίς στιγμιότυπο της KA21, ενώ η g μπορεί να κληθεί μόνο από στιγμιότυπο. Επομένως, η κλήση της g μέσα στην k παράγει λάθος μεταγλώττισης και η πρόταση Α είναι λανθασμένη. Ωστόσο, αυτό είναι το μοναδικό λάθος μεταγλώττισης στον κώδικα και η πρόταση Β είναι ορθή ενώ η Γ λανθασμένη. Αν διαγραφούν οι γραμμές που περιέχουν λάθη, δηλαδή η κλήση της g μέσα στην f, ο κώδικας θα τρέξει χωρίς να παραγάγει εξαίρεση. Στην main καλείται η k από μεταβλητή αναφοράς τύπου KA21 που έχει την τιμή null. Ωστόσο, η k είναι στατική και για την κλήση της δεν απαιτείται στιγμιότυπο παρά μόνο η κλάση στην οποία ανήκει. Καθώς ο τύπος της μεταβλητής kB είναι γνωστός, η επίλυση της κλήσης γίνεται με επιτυχία. Επομένως, η πρόταση ΣΤ είναι λανθασμένη. Εφόσον ο κώδικας θα τρέξει κανονικά με την αφαίρεση της g από την k, η έξοδός του είναι ff. Συνεπώς η πρόταση Δ είναι ορθή και η Ε λανθασμένη.

Μη στατικά μπλοκ αρχικοποίησης

Ποια η έξοδος του κώδικα που ακολουθεί;

```
class X {

    public static int i = 0;
}

public class KA22 {

    static X x1 = new X();
    static X x2 = new X();

    {
        System.out.print(x1.i);
```

```
    }

    public static void main(String[] args) {
        x1.i = 1;
        x2.i = 2;
        System.out.print(x1.i+””+x2.i);
    }
}
```

```
A.  012
B.  011
Γ.  012
Δ.  022
Δ.  12
E.  22
ΣΤ. 11
```

Απάντηση/Λύση

Η κλάση ΚΑ22 διαθέτει δύο στατικά μέλη, στιγμιότυπα της κλάσης Χ, τα x1 και x2. Κατά την εκτέλεση της main, στη μεταβλητή i της Χ εκχωρούνται διαδοχικά οι τιμές 1 και 2. Η i προσπελαύνεται μέσω των αναφορών x1 και x2, ωστόσο ως στατική μεταβλητή είναι κοινή για την κλάση Χ και ανεξάρτητη από τα στιγμιότυπα x1 και x2. Επομένως, μετά την εκχώρηση x2.i=2, η τιμή της i είναι 2 ανεξάρτητα αν την προσπελαύνουμε μέσω x1 ή x2. Το μπλοκ αρχικοποίησης πριν την main είναι μη στατικό και εκτελείται μόνο κατά τη δημιουργία στιγμιότυπων της ΚΑ22. Τέτοιο στιγμιότυπο όμως δεν δημιουργείται στον κώδικα, επομένως η print μέσα στο μπλοκ αρχικοποίησης δεν εκτελείται. Συνεπώς, η main εμφανίζει 22 και σωστή είναι η απάντηση Ε.

Μεταβλητές αναφοράς ως παράμετροι τιμής

Ποια η έξοδος του κώδικα που ακολουθεί;

```
public class KA23 {

    public static void f(StringBuilder s1, StringBuilder s2)
{
        s1 = new StringBuilder("f");
        s2.append(s1);
    }

    public static void main(String[] args) {
        StringBuilder s1 = new StringBuilder("s1"),
                s2 = new StringBuilder("s2");
        f(s1, s2);
        System.out.println(s1 + " " + s2);
    }
}
```

```
A. f s2f
B. f s2
Γ. s1 s2
Δ. s1 s2f
```

Απάντηση/Λύση

Τα αντικείμενα της κλάσης StringBuilder δεν είναι αμετάβλητα όπως τα αντικείμενα της String. Η append μέσα στην f θα προσθέσει στα περιεχόμενα της μνήμης όπου δείχνει η s2 τα περιεχόμενα της μνήμης στην

οποία αναφέρεται η s1. Δεδομένου ότι στην προηγούμενη γραμμή στην s1 έχει εκχωρηθεί η διεύθυνση στην οποία έχει τοποθετηθεί το f, το περιεχόμενο της s2 γίνεται s2f. Στην s1 όμως μέσα στην f εκχωρείται μια νέα διεύθυνση, δηλαδή αλλάζει η τιμή της s1 και όχι τα περιεχόμενα της μνήμης όπου αναφέρεται η s1. Καθώς η s1 είναι παράμετρος στην f, η μεταβολή αυτή ισχύει μόνο μέσα στην f, έξω από αυτήν, η s1 διατηρεί την αρχική τιμή της. Αυτή δείχνει σε θέση όπου έχει εκχωρηθεί η τιμή s1. Επομένως στην main εμφανίζεται s1 s2f και σωστή είναι η απάντηση Δ.

Μπλοκ αρχικοποίησης

Ποια η έξοδος του κώδικα που ακολουθεί;

```
public class KA24 {

    int i = 1;
    static int j = 2;

    {
        i = 3;
    }

    static {
        j = 4;
    }

    static {
        j = 5;
    }

    {
        i = 6;
    }

    KA24() {
        i = 7;
        j = 8;
    }

    public static void main(String[] args) {
        System.out.print(KA24.j);
        System.out.print(new KA24().i);
        System.out.print(KA24.j);
    }
}

A. 878
B. 578
Γ. 678
Δ. 567
```

Απάντηση/Λύση

Η πρώτη print αναφέρεται στη στατιτκή μεταβλητή j. Σε αυτήν τη φάση έχουν τρέξει οι αρχικοποιήσεις των στατικών μεταβλητών με τη σειρά που έχουν τοποθετηθεί στον κώδικα. Επομένως, η τιμή της j είναι 5. Στη δεύτερη print δημιουργείται ένα στιγμιότυπο, οπότε τρέχουν οι μη στατικές αρχικοποιήσεις και τελικά ο δημιουργός, οπότε οι μεταβλητές i και j λαμβάνουν την τιμή 7 και 8. Η έξοδος τους κώδικα είναι 578 και σωστή είναι η απάντηση Β.

Προκαθορισμένες μέθοδοι

Ποια η έξοδος του κώδικα που ακολουθεί;

```
interface I {
    public default int f(int i) {
        return 1;
    }
}
public class KA26 implements I {

    public String f() {
        return ("2");
    }
    public String f(int i) {
        return ("3");
    }

    public static void main(String[] args) {
        KA26 k=new KA26();
        System.out.println(k.f()+""+k.f(1));
    }
}
```

```
A. 23
B. 13
Γ. Λάθος χρόνου μεταγλώττισης
Δ. Λάθος χρόνου εκτέλεσης
```

Απάντηση/Λύση

Η μέθοδος f είναι προκαθορισμένη και όχι αφηρημένη στη διεπαφή I. Μπορεί βέβαια να επανοριστεί στην KA26. Όμως στην περίπτωση που επανοριστεί θα πρέπει να έχει τιμή επιστροφής συμβατή με την f της διεπαφής. Στην προκειμένη περίπτωση όμως έχουμε δύο f(int), δηλαδή δύο f με την ίδια ταυτότητα αλλά με διαφορετική τιμή επιστροφής. Έτσι αν ο κώδικας επιτρεπόταν, η κλάση KA26 θα διέθετε μία int f(int) που κληρονομεί από τη διεπαφή και μία String f(int). Σε αυτήν την περίπτωση όμως είναι αδύνατο να επιλυθεί μία κλήση του τύπου f(int). Επομένως, έχουμε λάθος μεταγλώττισης και σωστή είναι η απάντηση Γ. Σημειώστε πως η String f() δεν δημιουργεί πρόβλημα, καθώς εδώ έχουμε υπερφόρτωση και όχι επανορισμό.

Επανορισμός μεθόδων

Επιλέξτε όσες προτάσεις είναι αληθείς.

```
abstract class KA27A {

    private void f() {
        System.out.println("KA27A");
    }
}

abstract class KA27B {

    protected void f() {
        System.out.println("KA27B");
    }
```

```
}

class KA27C extends KA27A {

    public void f() {
        System.out.println("KA27C");
    }
}

class KA27 extends KA27B {

    @Override
    public void f() {
        System.out.println("KA27");
    }
}
```

Α. Η f στην ΚΑ27C επανορίζει την f στην ΚΑ27A
Β. Η f στην ΚΑ27C θα έπρεπε να σημειωθεί με την ετικέτα @Override
Γ. Η f στην ΚΑ27 επανορίζει την f στην ΚΑ27B
Δ. Η f στην ΚΑ27 προκαλεί λάθος μεταγλώττισης καθώς έχει δημόσια πρόσβαση ενώ η f στην γονική έχει προστατευμένη.

Απάντηση/Λύση

Η f στην ΚΑ27A έχει ιδιωτική πρόσβαση, δεν είναι προσβάσιμη στις παράγωγες και δεν μπορεί να επανοριστεί. Επομένως, η πρόταση Α είναι λανθασμένη. Η f στην ΚΑ27C είναι ανεξάρτητη από την f στην ΚΑ27A και δεν χρειάζεται την ετικέτα @Override. Αντίθετα, αν την προσθέσουμε, ο μεταγλωττιστής θα παράξει λάθος. Επομένως η πρόταση Β είναι λανθασμένη. Κατά τον επανορισμό μιας μεθόδου, είναι επιτρεπτό να αλλάξει ο προσδιοριστής πρόσβασης εφόσον ο προσδιοριστής στην παράγωγη είναι ευρύτερος από τον προσδιοριστή στη γονική. Πράγματι, εδώ έχουμε προστατευμένη πρόσβαση στη γονική και δημόσια στην παράγωγη. Επομένως, οι προτάσεις Γ και Δ είναι αληθείς.

Παραρτήματα

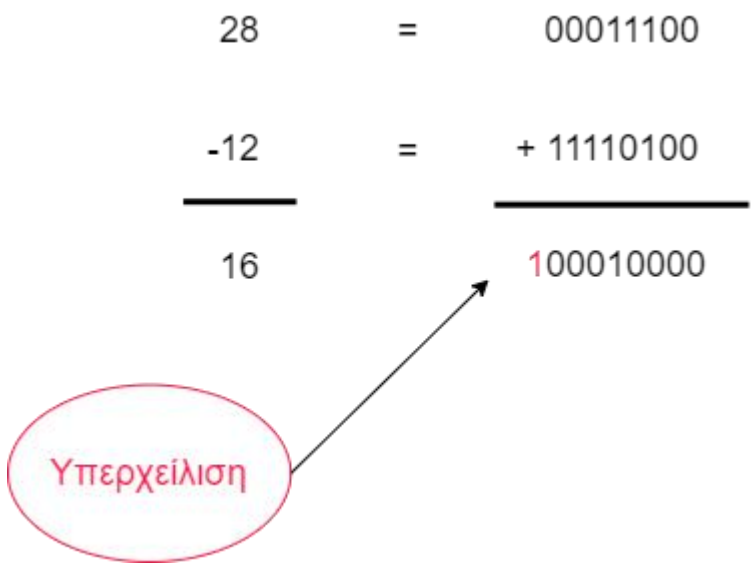
1. Κωδικοποίηση ακέραιων

Οι μη προσημασμένοι ακέραιοι κωδικοποιούνται ως ψηφιοχάρτες στη μνήμη εκφραζόμενοι απλώς στο δυαδικό σύστημα. Έτσι ένας μη προσημασμένος ακέραιος, π.χ. μήκους 8 δυαδικών ψηφίων, μπορεί να λάβει 28 διαφορετικές τιμές και πιο συγκεκριμένα τις τιμές από 0 έως 255. Ο μόνος μη προσημασμένος ακέραιος τύπος στην Java είναι ο τύπος char.

Αντίθετα, οι προσημασμένοι ακέραιοι χρησιμοποιούν το πρώτο bit για αποθήκευση του πρόσημου. Αν ο αριθμός είναι θετικός, το πρώτο bit είναι 0. Αν ο αριθμός είναι αρνητικός, το πρώτο bit είναι 1. Οι μεν θετικοί κωδικοποιούνται με βάση την παράσταση πρόσημο και μέτρο. Οι δε αρνητικοί με βάση την παράσταση συμπληρώματος ως προς 2 (two's complement). Πρόσημο και μέτρο σημαίνει πως στο πρώτο bit αποθηκεύεται το πρόσημο και στα υπόλοιπα το μέτρο στο δυαδικό σύστημα. Έτσι ένας προσημασμένος θετικός ακέραιος, σε λέξη μήκους 8 bits, έχει στη διάθεσή του 7 bits για αναπαράσταση του μέτρου του. Επομένως με 8 bits μπορούν να αναπαρασταθούν 27 διαφορετικές τιμές και συγκεκριμένα οι τιμές από 0 έως 27-1.

Ας δούμε τώρα πώς κωδικοποιείται ένας αρνητικός ακέραιος με ένα παράδειγμα. Έστω θέλουμε να κωδικοποιήσουμε το -12. Βρίσκουμε τη δυαδική αναπαράσταση του 12 που είναι 1100. Εφόσον διαθέτουμε 8 bits, η δυαδική αναπαράσταση γίνεται 00001100 καθώς είναι γνωστό πως η πρόσθεση μηδενικών στην αρχή ενός αριθμού δεν μεταβάλλει την τιμή του. Σε αυτήν την αναπαράσταση, αντιστρέφουμε όλα τα bits, οπότε παίρνουμε το συμπλήρωμα ως προς 1, δηλαδή τον ψηφιοχάρτη 11110011. Σε αυτόν τον δυαδικό αριθμό προσθέτουμε 1, οπότε λαμβάνουμε το συμπλήρωμα ως προς 2, δηλαδή τον 11110100. Αυτή η τελευταία αναπαράσταση είναι το -12 κωδικοποιημένο με το συμπλήρωμα ως προς 2.

Η κωδικοποίηση αυτή διευκολύνει πάρα πολύ στην πραγματοποίηση αφαιρέσεων με τα ίδια κυκλώματα που πραγματοποιούμε την πρόσθεση. Αν για παράδειγμα προσθέσουμε 28 + (-12), δηλαδή το 00011100 που είναι η αναπαράσταση του 28 με το 11110100 που είναι η αναπαράσταση του -12, τότε θα λάβουμε ως αποτέλεσμα το 100010000. Όμως η τιμή αυτή είναι μήκους 9 bits. Δεδομένου ότι η λέξη μας είναι μήκους 8 bits έχουμε υπερχείλιση του πρώτου bit, όπως φαίνεται στο σχήμα π.1, οπότε μένει ο ψηφιοχάρτης 00010000 που αναπαριστά πράγματι τη διαφορά 28-12=16.



Σχήμα π.1 *Αφαίρεση με κωδικοποίηση συμπλήρωμα ως προς 2*

2. Λέξεις κλειδιά της Java

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

Πίνακας π.1 *Λέξεις κλειδιά της Java*