Εισαγωγή στη Γλώσσα Προγραμματισμού Java

Ενότητα 4 – Αντικειμενοστρεφής Προγραμματισμός (Α' Μέρος)

4.1 Εισαγωγή

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

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

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

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

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

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

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

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

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

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

4.2 Αντικείμενο (Object)

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

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

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

4.3 Κλάση (Class)

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

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

Μία γλώσσα προγραμματισμού, για να θεωρείται αντικειμενοστρεφής θα πρέπει να υποστηρίζει κάποιες αρχές τις οποίες θα αναφέρουμε και θα αναλύσουμε ακολούθως. Η πρώτη αρχή είναι αυτή της ενθυλάκωσης (encapsulation), σύμφωνα με την οποία τα δεδομένα μιας κλάσης θα πρέπει να προστατεύονται από ανεξέλεγκτη πρόσβαση από άλλα σημεία του κώδικα. Τα δεδομένα είναι σαν να προστατεύονται μέσα σε ένα κουκούλι (ή κάψουλα), οπότε και προέκυψε ο όρος encapsulation. Έτσι λοιπόν, η πρόσβαση στα δεδομένα μιας κλάσης παρέχεται με ελεγχόμενο τρόπο, που ορίζει η ίδια η κλάση.
Μία επίσης σημαντική αρχή είναι αυτή της αφαιρετικότητας (abstraction), την οποία θίξαμε λίγο προηγουμένως αλλά τώρα θα αναλύσουμε λεπτομερώς. Με τον όρο αφαιρετικότητα αναφερόμαστε στις περιπτώσεις όπου αντιμετωπίζοντας κάποιο πολυσύνθετο πρόβλημα, επιλέγουμε να εξετάσουμε μόνο τις παραμέτρους του προβλήματος που μας αφορούν άμεσα, ενώ παράλληλα παραβλέπουμε σκοπίμως αυτές που δεν μας αφορούν. Για να το κατανοήσετε καλύτερα θα χρησιμοποιήσουμε το εξής παράδειγμα. Έστω πως θέλουμε να γράψουμε ένα πρόγραμμα για το Υπουργείο Συγκοινωνιών και μία από τις κλάσεις του προγράμματός μας θα είναι η **`Car`**, που όπως είναι προφανές από το όνομα αναπαριστά ένα αυτοκίνητο. Ένα αυτοκίνητο περιέχει έναν τεράστιο αριθμό από χαρακτηριστικά, από τον κατασκευαστή και το μοντέλο μέχρι το μέγεθος της βίδας του δισκόφρενου. Είναι προφανές πως αν κωδικοποιούσαμε όλα αυτά τα χαρακτηριστικά θα καταλήγαμε με μία κλάση τέρας, της οποίας μάλιστα τα 9/10 των χαρακτηριστικών θα μας ήταν παντελώς άχρηστα, δεδομένου πως στο Υπουργείο Συγκοινωνιών δεν ενδιαφέρονται για το μέγεθος της βίδας του δισκόφρενου ή για άλλα τέτοιου είδους κατασκευαστικά χαρακτηριστικά.
Τα στοιχεία ενός αυτοκινήτου που ενδιαφέρουν το Υπουργείο Συγκοινωνιών για να φέρει σε πέρας τη δουλειά του είναι για παράδειγμα, η μάρκα, το μοντέλο, ο κυβισμός, ο αριθμός πλαισίου, ο κυβισμός κλπ. Συγκεντρώνοντας όλα αυτά τα στοιχεία και κωδικοποιώντας τα, η κλάση μας θα περιείχε περίπου 10 μεταβλητές μέλη. Η συγκεκριμένη κλάση είναι σαφώς πιο κατάλληλη να χρησιμοποιηθεί για το συγκεκριμένο πρόβλημα από ότι η κλάση που περιέχει τα πάντα, αφού αφ'ενός περιέχει όλα τα στοιχεία που χρειαζόμαστε, εφ' ετέρου είναι σαφώς πιο εύχρηστη. Η έννοια της αφαιρετικότητας χρησιμοποιείται σε πολλούς διαφορετικούς τομείς, τόσο στις επιστήμες όσο και στην καθημερινή μας ζωή και βασίζεται ακριβώς σε αυτήν την αρχή, εξετάζω μόνο τις παραμέτρους του προβλήματος που με ενδιαφέρουν και παραβλέπω (αφαιρώ) αυτές που μου είναι αδιάφορες.
Η επόμενη αρχή είναι αυτή της απόκρυψης πληροφοριών (*information hiding*). Η συγκεκριμένη αρχή δεν είναι ιδιαίτερα σημαντική και αναφέρεται στην απόκρυψη των λεπτομερειών της υλοποίησης από τις χρήστες. Στη Java η συγκεκριμένη αρχή δεν υποστηρίζεται άμεσα.
Οι αρχές που ακολουθούν είναι αυτές που χαρίζουν στον αντικειμενοστρεφή προγραμματισμό το μεγαλύτερο μέρος της ισχύος του και θα τις αναλύσουμε και τις δύο διεξοδικά στην ενότητα 5. Μέσω της κληρονομικότητας (inheritance) μας παρέχεται η δυνατότητα επέκτασης μιας κλάσης και δημιουργίας μιας εξειδικευμένης παράγωγης με πρόσθετα στοιχεία. Από την άλλη πλευρά, ο

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

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

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

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

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

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

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

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

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

4.4 Το Πρόβλημα

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

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

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

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

4.5 Μεταβλητές Μέλη (Member-Variables)

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

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

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

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

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

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

4.6 Ορατότητα (Visibility)

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

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

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

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

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

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

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

Πίνακας 13

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

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

```
package elearning.geometry;

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

	// member variables

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

	// methods
}
```

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

4.7 Δημιουργία Αντικειμένων

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

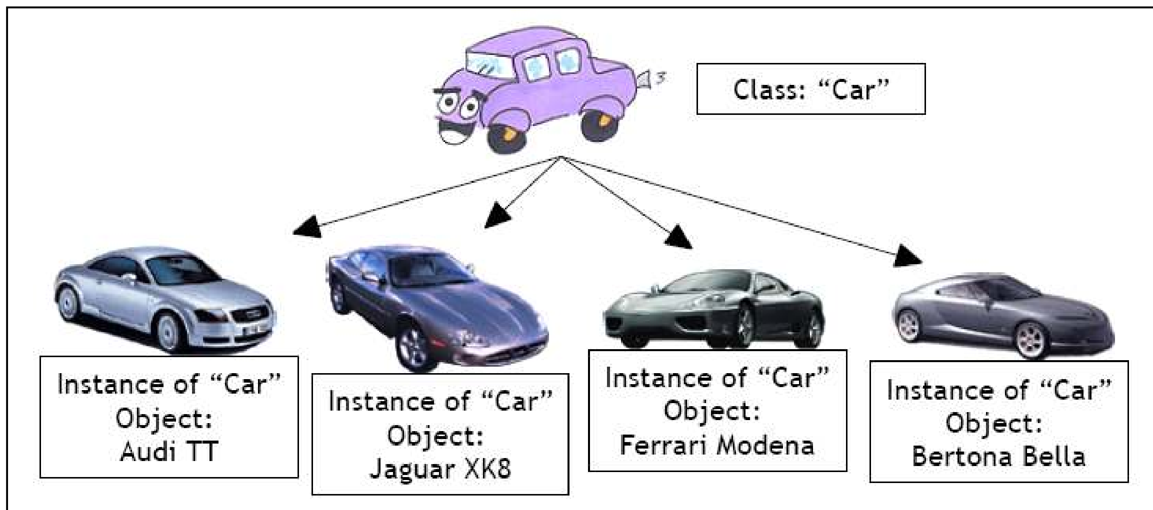
αντικείμενο κάποιας κλάσης, χρειαζόμαστε πρωτίστως μία αναφορά ιδίου τύπου με αυτόν της κλάσης από την οποία θέλουμε να δημιουργήσουμε αντικείμενο. Έτσι λοιπόν, αν η κλάση μας λεγόταν **MyClass** θα γράφαμε:

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

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

```
package elearning.geometry;

public class Main {

   public static void main(String[] args) {

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

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



Σχήμα 25

4.8 Μέθοδοι (Methods)

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

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

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

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

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

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

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

Μία μέθοδος έχει την ακόλουθη μορφή:

τι επιστρέφει | όνομα | τι όρισμα παίρνει

```
int   addNumbers   (int x, int y){ ... }
```

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

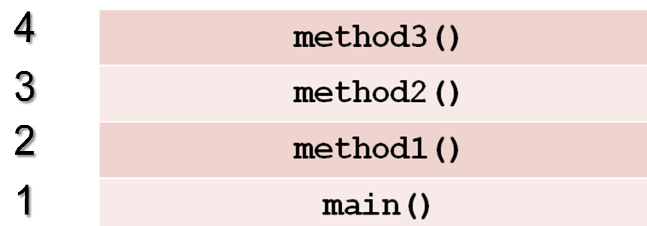
```
void doThis() { ... }
```

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

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

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

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



Σχήμα 26

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

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

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

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

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

Η πρώτη είναι η γνωστή μας πλέον **addNumbers()** που όμως τώρα βλέπουμε ολοκληρωμένη. Η μέθοδος αυτή λαμβάνει ως παραμέτρους δύο ακέραιους του οποίους προσθέτει μεταξύ τους και επιστρέφει το άθροισμα. Η δεύτερη ονομάζεται **displayGreeting()** και κάθε φορά που θα κληθεί θα εμφανίσει στην κονσόλα το μήνυμα **"Hello!"**.

Ένα άλλο σημείο στο οποίο θα πρέπει να εστιάσουμε είναι αυτό της χρήσης παραμέτρων μιας και έχει παρατηρηθεί πως το συγκεκριμένο θέμα προκαλεί δυσκολίες στους αρχάριους να το κατανοήσουν. Για τον λόγο αυτόν, θα χρησιμοποιήσουμε κάποια παραδείγματα ώστε να σας γίνει όσο το δυνατόν ξεκάθαρο. Μια μέθοδος λοιπόν, μπορεί να λαμβάνει παραμέτρους ή όχι. Οι παράμετροι είναι απαραίτητες όταν μια μέθοδος χρειάζεται έξτρα πληροφορίες για να επιλύσει το πρόβλημα που χειρίζεται. Φανταστείτε τις παραμέτρους δηλαδή ως τον τρόπο ανταλλαγής πληροφοριών μιας μεθόδου με τον εξωτερικό κόσμο. Για να γίνει καλύτερα κατανοητό, υποθέστε πως θέλουμε να γράψουμε μία μέθοδο η οποία υπολογίζει και επιστρέφει το εμβαδό ενός κύκλου και έχουμε δύο διαφορετικά σενάρια. Στην πρώτη εκδοχή, θα την τοποθετήσουμε σε μία γενική κλάση **Helper** που περιέχει βοηθητικές μαθηματικές μεθόδους και θα την δηλώσουμε ως **static**, ενώ στη δεύτερη εκδοχή θα την τοποθετήσουμε μέσα σε μία κλάση **Circle** που αναπαριστά έναν κύκλο.

Ξεκινώντας από την πρώτη περίπτωση, χρειαζόμαστε να γράψουμε μια μέθοδο που υπολογίζει και επιστρέφει το εμβαδό ενός κύκλου, η οποία θα λειτουργεί για κάθε κύκλο. Ο τύπος υπολογισμού του εμβαδού ενός κύκλου είναι όπως είδαμε και νωρίτερα ο πr^2. Πριν κωδικοποιήσουμε τον τύπο αυτόν σε μία συνάρτηση θα πρέπει να αναρωτηθούμε ποιος όρος του παραπάνω τύπου είναι αυτός που θεωρείται άγνωστος, ή καλύτερα ποιος όρος αλλάζει ανάλογα με τον κύκλο του οποίου το εμβαδό θέλουμε να υπολογίσουμε κάθε φορά. Το π παραμένει σταθερό και είναι γνωστό, μάλιστα υπάρχει ορισμένο στη Java ως σταθερά στην κλάση **Math**. Είναι προφανές πως ο όρος που μεταβάλλεται είναι η ακτίνα r. Ο συγκεκριμένος όρος όμως είναι απαράιτητος για τη μέθοδό μας ώστε να μπορέσει να υπολογίσει κάθε φορά το σωστό εμβαδό. Πως γνωρίζει η μέθοδος τίνος κύκλου ή καλύτερα βάσει ποιας ακτίνας θα πρέπει να υπολογίσει το εμβαδό; Σε τέτοιες περιπτώσεις λοιπόν, που μία μέθοδος χρειάζεται μία πληροφορία από το έξω πρόγραμμα, την κωδικοποιούμε με τη μορφή παραμέτρου.

Έτσι λοιπόν, η μέθοδός μας θα είχε την ακόλουθη μορφή:

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

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

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

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

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

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

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

Η πρώτη γραμμή θα χρησιμοποιούσε για να υπολογίσει το εμβαδό τη μεταβλητή μέλος του αντικειμένου **c1**, ενώ η δεύτερη τη μεταβλητή μέλος του αντικειμένου **c2**. Είναι πάρα πολύ σημαντικό να μπορέσετε να κατανοήσετε τη διαφορά μεταξύ των περιπτώσεων αυτών, πότε δηλαδή χρειαζόμαστε κάποια παράμετρο και πότε όχι επειδή την πληροφορία την έχουμε εσωτερικά από το ίδιο το αντικείμενο.

Κατά την εκτέλεση μιας μεθόδου, οι τυχόν παράμετροι που λαμβάνει συμπεριφέρονται ως τοπικές μεταβλητές αυτής, δηλαδή σαν να είχαν δηλωθεί τοπικά στο σώμα της μεθόδου. Η μέθοδος τις χρησιμοποιεί για να λύσει το πρόβλημα που χειρίζεται και όταν ολοκληρωθεί, αυτές διαγράφονται από τη μνήμη. Επίσης, θα πρέπει να τονιστεί πως κάθε μέθοδος εκτελείται σε ξεχωριστό χώρο μνήμης και αυτό σημαίνει πως τα ονόματα των παραμέτρων ή των τοπικών μεταβλητών μιας μεθόδου δεν έρχονται σε σύγκρουση με τα ονόματα αυτών που χρησιμοποιούνται από άλλες μεθόδους.

Τέλος, οι μέθοδοι μπορούν να δεχτούν ως παραμέτρους όλους τους βασικούς τύπους που γνωρίζουμε (π.χ. **int**, **double**, **byte** κλπ), καθώς επίσης αναφορές παντός τύπους όπως και αναφορές σε πίνακες.

4.9 Κλήση Κατ’ Αξία (Call By Value)

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

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

```
package elearning;

public class ByValue {

   public static void main(String[] args) {

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

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

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

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

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

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

4.10 Κλήση Κατ' Αναφορά (Call By Reference)

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

```
package elearning;

public class ByReference {

   public static void main(String[] args) {

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

      // call boo passing array
      boo(array);

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

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

Εκτελώντας τον παραπάνω κώδικα θα πάρετε την έξοδο που ακολουθεί:

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

Στο πρόγραμμα αυτό δηλώνεται ένας πίνακας ακεραίων 5 στοιχείων τα οποία αρχικοποιούνται με τις τιμές από το 1 έως το 5. Στη συνέχεια με τη βοήθεια μιας enhanced **for** εμφανίζουμε τις τιμές των στοιχείων του πίνακα. Αμέσως μετά υπάρχει η κλήση της **boo()**, η οποία λαμβάνει ως παράμετρο μια αναφορά σε πίνακα ακεραίων. Στην κλήση περνάμε ως παράμετρο την αναφορά **array** που δείχνει στον πίνακά μας. Στο σημείο αυτό ο compiler θα δημιουργήσει ένα αντίγραφο της **array** με την ίδια τιμή (άρα θα υπάρχουν δύο αναφορές που θα δείχνουν στον ίδιο πίνακα, η πρωτότυπη και το αντίγραφο) το οποίο και θα περάσει στην **boo()**.
Η **boo()** περιέχει μία **for** η οποία χρησιμοποιώντας την παράμετρο προσπελαύνει ένα ένα τα στοιχεία του πίνακα και τους δίνει νέα τιμή, συγκεκριμένα τις τιμές των ζυγών ακεραίων από το 2 έως το 10. Τερματίζοντας, η ροή επιστρέφει στη **main** όπου εμφανίζονται ξανά οι τιμές των στοιχείων του πίνακα στην κονσόλα με τη βοήθεια μιας enhanced **for**. Παρατηρήστε από την έξοδο πως οι τιμές του πίνακα έχουν αλλάξει και εμφανίζονται οι νέες στην κονσόλα.
Θα πρέπει να θυμάστε λοιπόν πως όταν έχουμε κλήση συνάρτησης που περιέχει αναφορές, τότε λειτουργεί ο μηχανισμός της κλήσης κατ' αναφορά. Ο συγκεκριμένος μηχανισμός είναι ιδιαίτερα χρήσιμος μιας και ο compiler δε σπαταλάει χρόνο και μνήμη για να κατασκευάσει αντίγραφο ολόκληρου του αντικειμένου όπως για παράδειγμα συμβαίνει στη C++, όπου για να επιτευχθεί αντίστοιχη λειτουργία θα πρέπει να ενεργήσει κατάλληλα ο προγραμματιστής.

4.11 Υπερφόρτωση Μεθόδων (Method Overloading)

Η Java όπως όλες οι σύγχρονες αντικειμενοστρεφείς γλώσσες υποστηρίζει τη δυνατότητα υπερφόρτωσης μεθόδων. Αυτό σημαίνει, πως μπορούμε να έχουμε μεθόδους στην ίδια κλάση (ή σε υποκλάσεις της) που έχουν το ίδιο όνομα και διαφοροποιούνται μεταξύ τους ως προς τις παραμέτρους που δέχονται (αριθμό ή/και τύπο) και ίσως τον τύπο επιστροφής.
Για να καταλάβετε γιατί η συγκεκριμένη δυνατότητα είναι χρήσιμη, υποθέστε πως έχουμε το εξής πρόβλημα. Θέλουμε να γράψουμε μία μέθοδο που θα λαμβάνει ως παραμέτρους δύο αριθμούς και θα επιστρέφει τον μεγαλύτερο από τους δύο. Επιπλέον, θα πρέπει η μέθοδος αυτή να λειτουργεί για όλους τους βασικούς τύπους πλην του **char** και του **boolean**.
Αν ξεκινούσατε να γράψετε τον κώδικα που λύνει το συγκεκριμένο πρόβλημα με βάση αυτά που γνωρίζετε μέχρι τώρα, σίγουρα θα καταλήγατε με 6 μεθόδους με πανομοιότυπο (αν όχι τον ίδιο) κώδικα, κάθε μία από τις οποίες θα έπρεπε να έχει και ξεχωριστό όνομα. Για παράδειγμα οι μέθοδοι για τους τύπους **int** και **double** θα ήταν οι ακόλουθες:

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

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

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

```
double maxDouble(double a, double b)
          └────── method signature ──────┘
```

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

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

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

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

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

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

4.12 Δημιουργοί (Constructors)

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

Όπως αναφέρθηκε ήδη σε προηγούμενη υποενότητα, μία από τις κατηγορίες μεθόδων που περιέχονται σε μία κλάση είναι οι διαχειριστικές, που παίζουν σημαντικό ρόλο κατά τη διάρκεια ζωής του αντικειμένου. Το πιο χαρακτηριστικό είδος μεθόδων της κατηγορίας αυτής και ίσως οι πιο σημαντικές είναι οι δημιουργοί.
Ένας δημιουργός (constructor) είναι μία ειδική μέθοδος η οποία είναι υπεύθυνη για τη δημιουργία των αντικειμένων μιας κλάσης και για τον λόγο αυτόν, κάθε κλάση θα πρέπει να περιέχει τουλάχιστον έναν. Οι constructors έχουν κάποια χαρακτηριστικά που τους κάνουν να ξεχωρίζουν και οπτικά από τις άλλες μεθόδους. Τα χαρακτηριστικά αυτά είναι:

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

Επιπλέον, ένας constructor μπορεί όπως οι κοινές μέθοδοι να λαμβάνει παραμέτρους ή όχι, ενώ το γεγονός πως μπορούμε να έχουμε περισσότερους του ενός σε μία κλάση σημαίνει πως μπορούν να υπερφορτωθούν. Πράγματι, οι constructors αποτελούν χαρακτηριστικό παράδειγμα μεθόδων που υπερφορτώνονται, αφού συνήθως σε μία κλάση έχουμε τουλάχιστον δύο.
Οι constructors είναι το μέρος που τοποθετούμε κώδικα αρχικοποίησης για το κάθε αντικείμενο, π.χ. για να αποδώσουμε αρχικές τιμές στο αντικείμενο (η πιο συνήθης χρήση), να κάνουμε δέσμευση πόρων, έλεγχο εγκυρότητας τιμών κλπ. Αν και οι constructors μπορούν να έχουν οποιονδήποτε προσδιοριστή ορατότητας, τους κάνουμε πάντοτε **public** όπως άλλωστε και τις άλλες μεθόδους της κλάσης. Δίνοντας προσδιοριστή ορατότητας **private** στους constructors μιας κλάσης, δε θα είναι δυνατόν να δημιουργηθούν αντικείμενα.
Δεδομένου πως η ύπαρξη ενός τουλάχιστον constructor είναι υποχρεωτική και ζωτικής σημασίας για οποιαδήποτε κλάση ώστε να μπορεί να δημιουργήσει αντικείμενα, η Java χρησιμοποιεί έναν μηχανισμό που εξασφαλίζει πως σε κάθε κλάση θα υπάρχει σίγουρα ένας. Σύμφωνα με τον μηχανισμό αυτόν λοιπόν, αν σε μία κλάση δεν ορίσει ο προγραμματιστής κάποιον constructor, ο compiler θα δημιουργήσει έναν από μόνος του, ακόμη κι αν στον κώδικα δε βλέπουμε τις αντίστοιχες γραμμές ορισμού του. Αυτόματα δηλαδή κατά το compilation ο compiler θα προσθέσει τις γραμμές που πρέπει, ανεξάρτητα αν σε εμάς αυτή η διαδικασία δεν είναι ορατή. Ο constructor που δημιουργεί ο compiler ονομάζεται default constructor (προκαθορισμένος) και δε λαμβάνει καμία παράμετρο. Επιπλέον, δεν περιέχει καθόλου κώδικα στο σώμα του.
Παρόλα αυτά, η λειτουργία του μηχανισμού αυτόματης παραγωγής του default constructor ακολουθεί και αυτή κάποιους κανόνες. Ποιο συγκεκριμένα, για να παράξει ο compiler από μόνος του τον default constructor θα πρέπει ο προγραμματιστής να μην έχει ορίσει κανέναν constructor στην κλάση του, είτε κάποιον που λαμβάνει παραμέτρους είτε όχι. Σε περίπτωση που έχει οριστεί από τον προγραμματιστή έστω και ένας, ο default constructor δε θα παραχθεί αυτόματα.
Στα παρακάτω αποσπάσματα κλάσεων για παράδειγμα, στη μία περίπτωση ο compiler θα παράξει αυτόματα τον default constructor ενώ στην άλλη όχι.

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

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

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

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

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

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

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

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

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

```
package elearning.geometry;

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

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

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

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

4.13 Καταστροφή Αντικειμένων

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

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

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

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

4.14 Getters/Setters

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

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

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

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

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

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

```
package elearning.geometry;

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

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

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

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

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

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

    public int getX(){
        return x;
    }

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

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

4.15 Στατικά Μέλη

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

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

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

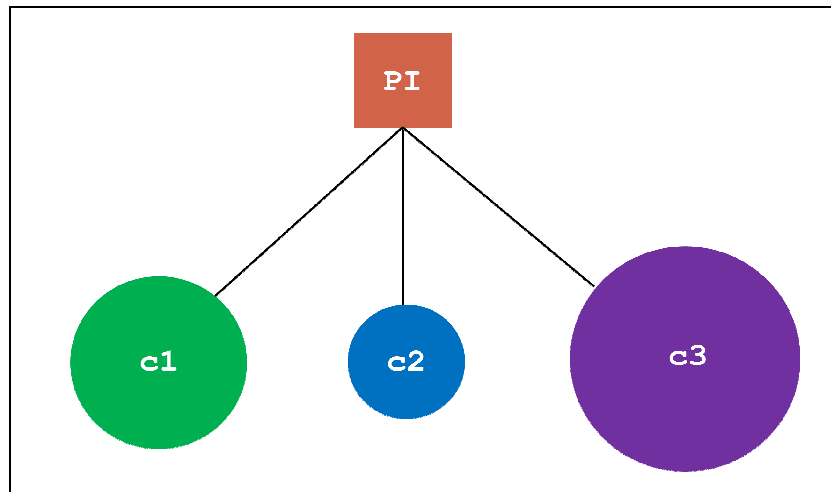
πως ο τύπος υπολογισμού του εμβαδού ενός κύκλου είναι ο πr^2, στον οποίο ο μοναδικός άγνωστος είναι κάθε φορά η ακτίνα. Δεδομένου πως η μέθοδος θα τοποθετηθεί στην κλάση που αναπαριστά κύκλους, είναι βέβαιο πως η ακτίνα θα έχει εκφραστεί στην κλάση αυτή ως μεταβλητή μέλος και άρα θα μπορούμε να την χρησιμοποιήσουμε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

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

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

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

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



Σχήμα 27

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

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

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

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

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

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

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

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

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

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

*<u>Σημείωση</u>: Οι **static** μέθοδοι μπορούν να προσπελάσουν <u>μόνο</u> **static** μεταβλητές ή σταθερές.*

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

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

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

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

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

4.16 Ολοκληρωμένη Λύση

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

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

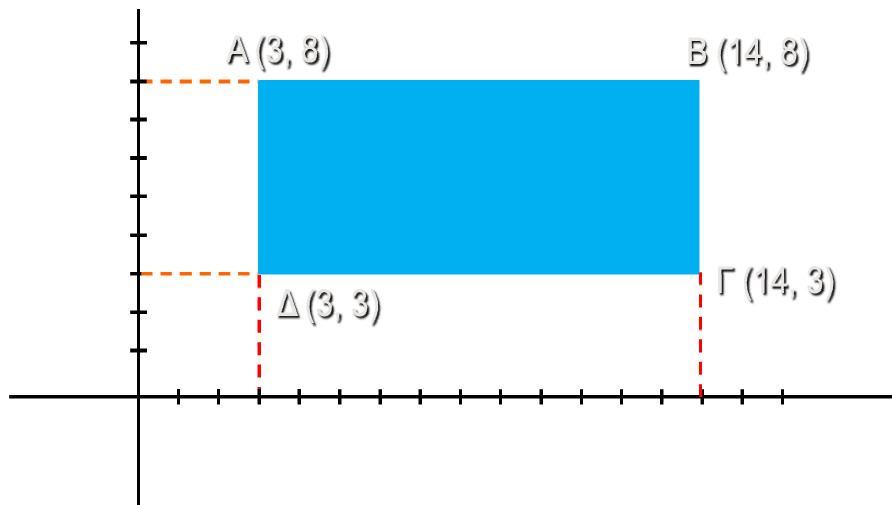
```
package elearning.geometry;

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

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

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

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

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

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

   public int getX(){
      return x;
   }

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

   public int getY(){
      return y;
   }

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

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

```
package elearning.geometry;

// class to represent a circle
public class Circle {

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

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

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

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

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

   public int getRadius() {
      return radius;
   }

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

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

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

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

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

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



Σχήμα 28

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

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

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

```
package elearning.geometry;

// class to represent a rectangle
public class Rectangle {

   // member variables
   private Point[] points;

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

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

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

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

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

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

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

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

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

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

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

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

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

```
package elearning.geometry;

public class Main {

   public static void main(String[] args) {

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

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

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

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

      // display circle data
      c1.displayCircleData();

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

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

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

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

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

Εκτελώντας το κεντρικό πρόγραμμα, θα πάρετε την ακόλουθη έξοδο:

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

4.17 Απαριθμητοί Τύποι (Enumerated Types)

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

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

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

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

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

```
package elearning;

public class EnumTest {
   Day day;

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

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

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

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

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

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

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

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