Εισαγωγή στη Γλώσσα Προγραμματισμού Java

Ενότητα 5 – Αντικειμενοστρεφής Προγραμματισμός (Β' Μέρος)

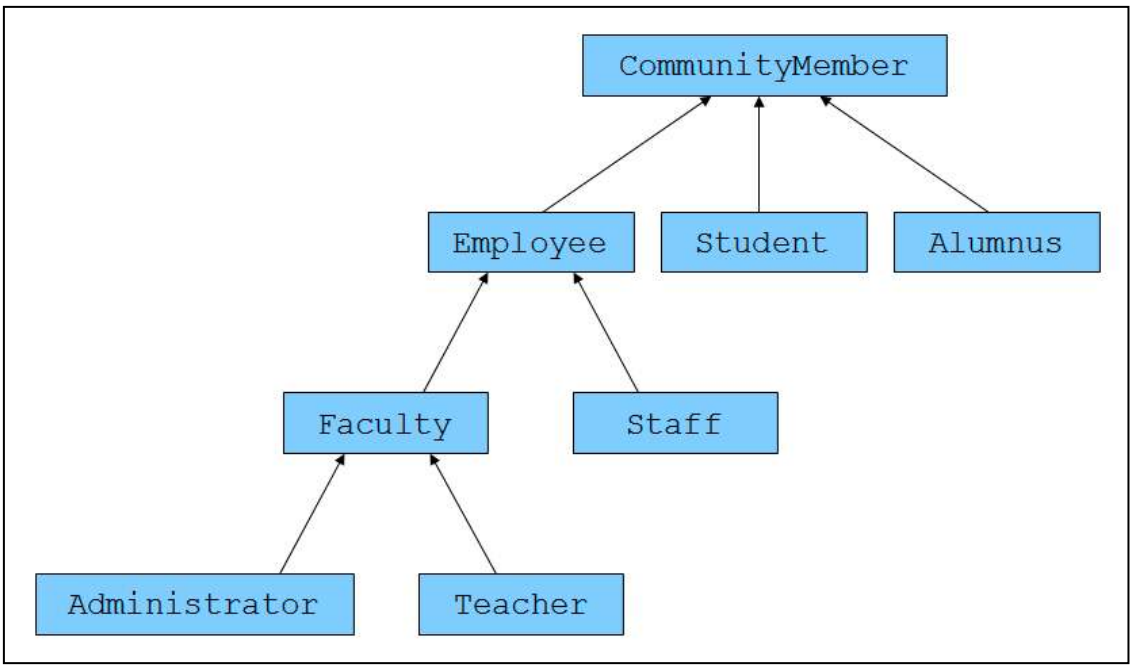
5.1 Κληρονομικότητα (Inheritance)

Η τρέχουσα ενότητα καλύπτει κάποιες πιο σημαντικές αρχές του αντικειμενοστρεφούς προγραμματισμού, την κληρονομικότητα και τον πολυμορφισμό. Ξεκινάμε με την αρχή της κληρονομικότητας, στην οποία όπως θα δούμε στη συνέχεια αφ' ενός βασίζεται ο σχεδιασμός και η υλοποίηση συστημάτων, αφ' ετέρου χρησιμοποιείται κατά κόρον και στην αρχιτεκτονική της ίδιας της Java.
Η αρχή της κληρονομικότητας αφορά στην δημιουργία μιας νέας κλάσης η οποία ονομάζεται παράγωγη (derived) από μία υπάρχουσα (ονομάζεται βασική ή base class) και άρα με τον τρόπο αυτόν επιτυγχάνεται επαναχρησιμοποίηση κώδικα. Ως αποτέλεσμα του μηχανισμού της κληρονομικότητας υπάρχουν οι εξής δυνατότητες:

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

Η λογική λοιπόν που ακολουθείται κατά τον σχεδιασμό και το "χτίσιμο" συστημάτων, είναι να ξεκινήσουμε από μία βασική κλάση (την οποία μπορεί να υλοποιήσουμε οι ίδιοι ή να προμηθευτούμε από κάποια άλλη πηγή) η οποία περιλαμβάνει τα απολύτως απαραίτητα στοιχεία που περιγράφουν ένα αντικείμενο, η μια κατηγορία αντικειμένων. Από την κλάση αυτή στη συνέχεια δημιουργούμε παράγωγες κλάσεις, που ανάλογα με τις παραμέτρους του προβλήματός μας, μπορεί να προσθέτουν χαρακτηριστικά ή/και συμπεριφορές.
Ως εκ τούτου, έχοντας ολοκληρώσει τη διαδικασία σχεδιασμού του συστήματός μας, καταλήγουμε να έχουμε μία ιεραρχία κλάσεων (class hierarchy), που έχει τη μορφή του σχήματος 29. Μία ιεραρχία κλάσεων είναι ένα "δέντρο" που διαβάζεται από επάνω προς τα κάτω. Στην κορυφή βρίσκεται η βασική κλάση, που όπως είπαμε περιλαμβάνει τα βασικά χαρακτηριστικά του ενός αντικειμένου, ενώ όσο προχωράμε προς τα κάτω οι κλάσεις εξειδικεύονται προσθέτοντας χαρακτηριστικά και συμπεριφορές, κατάλληλες για την επίλυση του προβλήματος. Κάθε κλάση στην ιεραρχία κληρονομεί χαρακτηριστικά από αυτές που βρίσκονται επάνω από αυτήν ενώ παράλληλα κληροδοτεί σε αυτές που βρίσκονται από κάτω της.
Στο σχήμα 29 απεικονίζεται μία ιεραρχία κλάσεων που θα μπορούσε να χρησιμοποιηθεί σε ένα πρόγραμμα διαχείρισης του προσωπικού ενός πανεπιστημίου. Η βασική κλάση ονομάζεται **`CommunityMember`**, δηλαδή μέλος της πανεπιστημιακής κοινότητας και θα μπορούσε να περιλαμβάνει ως μεταβλητές μέλη για παράδειγμα το ονοματεπώνυμο, το φύλο, την ημερομηνία γέννησης κλπ.
Από την **`CommunityMember`** κληρονομούν οι **`Employee`**, **`Student`** και **`Alumnus`**. Η πρώτη αντιπροσωπεύει τους εργαζόμενους στο πανεπιστήμιο, η δεύτερη αντιπροσωπεύει τους φοιτητές ενώ

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



Σχήμα 29

Από την κλάση **`Employee`** κληρονομούν οι **`Faculty`** και **`Staff`**, η πρώτη αναπαριστά τους εργαζόμενους που παράλληλα είναι μέλη κάποιου τμήματος, ενώ η δεύτερη τους απλούς υπαλλήλους. Η **`Faculty`** θα μπορούσε να προσθέσει το τμήμα στο οποίο ανήκει ο εργαζόμενος, ενώ η **`Staff`** το αντικείμενο εργασίας του.
Τέλος, από την **`Faculty`** κληρονομούν οι **`Administrator`** και **`Teacher`**, η μεν πρώτη αντιπροσωπεύει τους εργαζόμενους με διαχειριστικό ρόλο (π.χ. γραμματείς κλπ), η δε δεύτερη τους διδάσκοντες. Βλέπουμε λοιπόν πως ξεκινώντας από μία απλή βασική κλάση και χρησιμοποιώντας την αρχή της κληρονομικότητας καταλήγουμε να έχουμε δημιουργήσει κλάσεις που καλύπτουν όλο το φάσμα των μελών μιας επιστημονικής κοινότητας, με την απαιτούμενη λεπτομέρεια.

5.2 Σχέσεις “IS-A”, “HAS-A”

Κατά το σχεδιασμό συστημάτων χρησιμοποιώντας το αντικειμενοστρεφές μοντέλο, προκύπτουν κάποιες σχέσεις μεταξύ των κλάσεων. Οι πιο βασικές από αυτές είναι η σχέση “IS-A” (είναι ένα) και η σχέση “HAS-A” (έχει ένα).
Η σχέση “IS-A” ισχύει πάντοτε μεταξύ δύο κλάσεων **`A`** και **`B`** όπου η μία κληρονομεί από την άλλη είτε άμεσα (η **`B`** άμεση υποκλάση της **`A`**), είτε έμμεσα (η **`B`** κληρονομεί από μία κλάση **`X`** η οποία έχει κληρονομήσει από την **`A`**). Το συγκεκριμένο γεγονός, ότι δηλαδή ισχύει μία σχέση “IS-A” μεταξύ μιας παράγωγης κλάσης και της μητρικής της, είναι και ένα από τα δυνατά σημεία της κληρονομικότητας μιας και μας δίνει τη δυνατότητα να χειριζόμαστε ένα αντικείμενο της παράγωγης κλάσης και ως αντικείμενο της κλάσης βάσης.

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

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

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

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

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

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

5.3 Υλοποίηση Κληρονομικότητας

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

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

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

Στην Java ισχύει ένας πολύ βασικός κανόνας σύμφωνα με τον οποίο η οποιαδήποτε κλάση, θα πρέπει να κληρονομεί οπωσδήποτε από κάποια άλλη. Αυτό αρχικά θα σας φανεί παράξενο, αφού μέχρι στιγμής έχετε δημιουργήσει κάποιες απλές κλάσεις και ποτέ δε χρησιμοποιήσατε κληρονομικότητα. Αυτό που συμβαίνει είναι πως όταν δημιουργούμε μία νέα κλάση, στην ουσία έχουμε δύο επιλογές. Η μία είναι να κληρονομήσουμε άμεσα από μία κλάση βάσης, όπως μάθαμε στην τρέχουσα ενότητα. Κάνοντας αυτό, ο κανόνας που μόλις αναφέραμε δεν παραβιάζεται μιας και κληρονομούμε άμεσα από μία άλλη κλάση. Η δεύτερη επιλογή που συμβαίνει στις περισσότερες περιπτώσεις, είναι αν η κλάση που δημιουργούμε δεν κληρονομεί άμεσα από κάποια άλλη, ο compiler θα την θέσει αυτόματα να κληρονομήσει από την κλάση **`Object`**. Αυτό συμβαίνει πάντοτε όταν μία κλάση δεν κληρονομεί άμεσα από κάποια άλλη, ακόμη κι αν δεν το βλέπετε γραμμένο ρητά στον κώδικα (δεν θα δείτε ποτέ π.χ. statement όπως **`public class Car extends Object`**). Άρα λοιπόν, και στην περίπτωση αυτή ο κανόνας δεν παραβιάζεται, μιας και η κλάση μας θα κληρονομήσει από την **`Object`**.
Η κλάση **`Object`** που θα την μελετήσουμε αναλυτικότερα αργότερα, είναι στην ουσία η υπερκλάση όλων των κλάσεων που υπάρχουν ή που μπορεί να δημιουργηθούν στη Java. Τώρα σας είναι πλέον ξεκάθαρο τι εννοούσαμε όταν σας είπαμε νωρίτερα πως η αρχή της κληρονομικότητας χρησιμοποιείται κατά κόρον και από την ίδια την αρχιτεκτονική της γλώσσας.
Ας δούμε όμως τώρα τους κανόνες που ισχύουν κατά την κληροδότηση. Στην περίπτωση λοιπόν που μία κλάση κληρονομεί από κάποια άλλη, τότε ισχύουν τα εξής:

- Η παράγωγη κλάση κληρονομεί όλα τα **`public`** μέλη της κλάσης βάσης
- Η παράγωγη κλάση έχει άμεση πρόσβαση σε όλα τα **`public`** και **`protected`** μέλη της κλάσης βάσης, καθώς και στα *default* αν βρίσκεται στο ίδιο πακέτο με αυτήν
- Τα **`private`** μέλη της κλάσης δεν κληρονομούνται άμεσα, αλλά έχουμε πρόσβαση σε αυτά μέσω των **`public`** μεθόδων της κλάσης βάσης

Για να προσπελάσουμε μία μεταβλητή μέλος της κλάσης βάσης από την υποκλάση, χρησιμοποιούμε τη σύνταξη:

```
super.baseclass_variable π.χ.
super.colour = "blue";
```

5.4 Δημιουργία/Καταστροφή Αντικειμένων Παραγώγων Κλάσεων

Στην προηγούμενη υποενότητα είδαμε πως η παράγωγη κλάση κληρονομεί όλα τα **`public`** μέλη της μητρικής. Αυτό ισχύει για όλα τα μέλη πλην των constructors. Συγκεκριμένα, οι constructors είναι τα μόνα δομικά στοιχεία μιας κλάσης που δεν κληρονομούνται και συνεπώς, κάθε κλάση θα πρέπει να

ορίζει τους δικούς της. Στο σημείο αυτό υπάρχει άλλος ένας κανόνας της Java, σύμφωνα με τον οποίο το πρώτο statement στον constructor μιας υποκλάσης θα πρέπει να είναι μία κλήση στον constructor της υπερκλάσης π.χ.

```
public Volleyball() {
   super();   // parent class constructor call
   ...
}
```

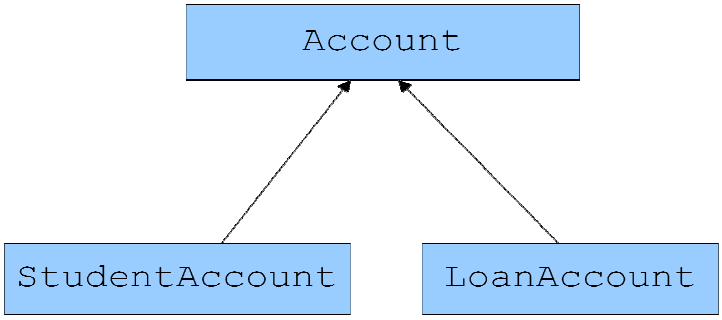
Μάλιστα, αν αυτό το statement δε γραφτεί άμεσα από εμάς, θα εισαχθεί αυτόματα από τον compiler. Ας δούμε όμως γιατί υπάρχει ο συγκεκριμένος κανόνας. Κατά τη δημιουργία ενός αντικειμένου παράγωγης κλάσης, ο constructor της παράγωγης καλεί αυτόν της κλάσης βάσης. Θυμηθείτε λίγο τη στοίβα κλήσεων μεθόδων που συζητήσαμε στην προηγούμενη ενότητα. Έχοντας καλέσει τον constructor της κλάσης βάσης, ο constructor της παράγωγης κλάσης περιμένει στον πάτο της στοίβας. Αν τώρα υποθέσουμε πως έχουμε μία ιεραρχία που αποτελείται από τέσσερα επίπεδα, η ίδια διαδικασία θα εξακολουθήσει να εκτελείται (ο constructor της παράγωγης καλεί τον constructor της κλάσης βάσης της και περιμένει) μέχρις ότου φτάσουμε στην κορυφή της ιεραρχίας, όπου βρίσκεται η κλάση **`Object`**. Ο constructor της **`Object`** δηλαδή είναι ο τελευταίος που καλείται και ο πρώτος που ολοκληρώνει την εκτέλεσή του. Έτσι θα δημιουργηθεί ένα αντικείμενο της κλάσης **`Object`** και στη συνέχεια θα ολοκληρωθεί η κλήση του constructor της κλάσης που κληρονομεί άμεσα από αυτήν. Έχουμε δηλαδή την αντίστροφη διαδικασία από πριν, με τους constructors να ολοκληρώνουν και να δημιουργούν αντικείμενα της κάθε κλάσης, μέχρις ότου φτάσουμε στην παράγωγη κλάση που βρίσκεται στον πάτο της ιεραρχίας, από όπου εκκίνησε η όλη διαδικασία. Όταν ολοκληρωθεί και ο τελευταίος constructor, θα βρίσκεται στη μνήμη ένα αντικείμενο από κάθε κλάση της ιεραρχίας.

Κανόνας σωστής πρακτικής: Αν και όπως ειπώθηκε ο compiler θα εισάγει από μόνος του μία κλήση στον default constructor της μητρικής κλάσης σε περίπτωση που το statement απουσιάζει, είναι καλύτερα να μην εξαρτάστε από αυτόν και να γράψετε άμεσα από μόνοι σας την κατάλληλη κλήση στον constructor που επιθυμείτε.

Κατά την καταστροφή τώρα ενός αντικειμένου παράγωγης κλάσης, ακολουθείται η αντίστροφη σειρά. Όταν ένα αντικείμενο πρόκειται να καταστραφεί, πριν από τη διαγραφή του από τη μνήμη καλείται αυτόματα μία μέθοδος με το όνομα **`finalize()`**, την οποία κληρονομούν όλες οι κλάσεις από την **`Object`**. Η συγκεκριμένη μέθοδος είναι χρήσιμη όταν για παράδειγμα έχουμε δεσμεύσει κάποιους πόρους κατά τη δημιουργία του αντικειμένου μέσω ενός constructor. Αυτοί οι πόροι θα πρέπει να αποδεσμευτούν πριν την καταστροφή του αντικειμένου και η **`finalize()`** είναι το κατάλληλο μέρος για να γίνει αυτό. Σε τέτοιες περιπτώσεις λοιπόν, υπερκαλύπτουμε την **`finalize()`** (θα μιλήσουμε για την υπερκάλυψη μεθόδων στην υποενότητα 5.7) και γράφουμε μέσα της τον κώδικα αποδέσμευσης των πόρων.
Κατά την καταστροφή λοιπόν ενός αντικειμένου παράγωγης κλάσης, καλείται η **`finalize()`** της παράγωγης κλάσης και διαγράφεται το αντικείμενο από τη μνήμη, στη συνέχεια καλείται η **`finalize()`** της άμεσα μητρικής κλάσης του οπότε και αυτό διαγράφεται από τη μνήμη και η διαδικασία αυτή συνεχίζεται με κατεύθυνση προς τα επάνω μέχρις ότου φτάσει στην κλάση **`Object`** της οποίας η **`finalize()`** είναι και η τελευταία που θα κληθεί. Τελευταίο αντικείμενο που θα διαγραφτεί από τη μνήμη είναι το αντικείμενο τύπου **`Object`**.

5.5 Ορατότητα Παράγωγης Κλάσης

Όπως έχει ήδη αναφερθεί, η κληρονομικότητα είναι μία από τις πιο βασικές αρχές του αντικειμενοστρεφούς προγραμματισμού, στην οποία βασίζεται ο σχεδιασμός και η υλοποίηση των σύγχρονων συστημάτων. Παρόλα αυτά, δεν είναι λίγες οι φορές που τα συστήματα αυτά πάσχουν από προβλήματα που αποτελούν συνέπεια της όχι σωστής σχεδίασης. Στην τρέχουσα υποενότητα θα εξετάσουμε ένα παράδειγμα το οποίο θα λύσουμε με διαφορετικούς τρόπους, ώστε να επιδείξουμε κάποια από τα κοινά λάθη που μπορούμε να κάνουμε κατά τη σχεδίαση κυρίως κάνοντας χρήση της κληρονομικότητας καθώς επίσης θα σας παρουσιαστεί και η λύση η οποία βασίζεται στους κανόνες σωστής πρακτικής και θεωρείται η πιο ενδεδειγμένη.
Ας υποθέσουμε λοιπόν πως έχουμε να υλοποιήσουμε μία εφαρμογή για μία τράπεζα. Ένα μικρό τμήμα της ιεραρχίας είναι αυτό που φαίνεται στο σχήμα 30 και που αναπαριστά κάποια από τα προϊόντα της τράπεζας. Στην κορυφή της ιεραρχία βρίσκεται η κλάση **Account**, από την οποία κληρονομούν οι κλάσεις **StudentAccount** που αναπαριστά έναν φοιτητικό λογαριασμό και η **LoanAccount** που αναπαριστά έναν δανειακό λογαριασμό.



Σχήμα 30

Για να κρατήσουμε το παράδειγμα απλό, θα θεωρήσουμε πως τα στοιχεία που αποθηκεύονται ως μεταβλητές μέλη στη μητρική κλάση και που είναι κοινά για όλα τα είδη λογαριασμών είναι μόνο το ονοματεπώνυμο του δικαιούχου, το επιτόκιο και το υπόλοιπο. Ένας φοιτητικός λογαριασμός έχει επιπλέον τη δυνατότητα υπερανάληψης η οποία θα προστεθεί ως μεταβλητή μέλος, ενώ ένας δανειακός θα προθέσει ως μεταβλητή μέλος τη μηνιαία δόση που θα πρέπει να καταβάλλεται.
Ξεκινώντας την υλοποίηση της **Account** με βάση τα όσα έχουμε μάθει μέχρι τώρα, θα καταλήγαμε με τον παρακάτω κώδικα:

```
package elearning;

public class Account {
   // instance variables
   private String holder;
   private double balance;
   private double interest;

   // methods
   public Account(){}

   public Account(String h, double b, double i){
      holder = h;
```

```
        balance = b;
        interest = i;
    }

    public String getHolder() {
        return holder;
    }

    public void setHolder(String h) {
        holder = h;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double b) {
        balance = b;
    }

    public double getInterest() {
        return interest;
    }

    public void setInterest(double i) {
        interest = i;
    }
}
```

Ο κώδικας της **Account** είναι απλός. Η κλάση περιέχει δύο constructors, τον default συν έναν που αρχικοποιεί όλες τις μεταβλητές μέλη του αντικειμένου που δημιουργείται και τις κατάλληλες getters/setters.
Στη συνέχεια θα ξεκινούσαμε την υλοποίηση της **StudentAccount**. Ο κώδικάς της είναι ο ακόλουθος:

```
package elearning;

public class StudentAccount extends Account {
    // instance variables
    private double overdraft;

    // methods
    StudentAccount(){}

    StudentAccount(String h, double b, double i, double o){
        super.holder = h;
        super.balance = b;
        super.interest = i;
        overdraft = o;
    }

    public double getOverdraft() {
        return overdraft;
    }

    public void setOverdraft(double o) {
        overdraft = o;
    }
}
```

Αν πληκτρολογήσετε τον κώδικα της κλάσης **StudentAccount**, θα δείτε πως δε θα κάνει compile. Το πρόβλημα υπάρχει στον constructor που αρχικοποιεί το αντικείμενο με τιμές. Ας τον δούμε όμως πιο προσεκτικά μιας και υπάρχουν πολλά θέματα προς συζήτηση. Κατ' αρχήν, παρατηρήστε πως δεν έχουμε γράψει ρητά την κλήση στον constructor της μητρικής κλάσης στην πρώτη γραμμή, που σημαίνει πως ο compiler θα καλέσει τον default constructor εισάγοντας το statement **super()** αυτόματα και ο κώδικας θα λειτουργήσει κανονικά χωρίς πρόβλημα. Επίσης θα παρατηρήσατε πως ο συγκεκριμένος constructor λαμβάνει τέσσερις παραμέτρους, ενώ η ίδια η κλάση διαθέτει μόνο μία μεταβλητή μέλος. Για ποιο λόγο υπάρχουν τέσσερις παράμετροι; Η απάντηση είναι πως εκτός της μίας αυτής παραμέτρου, υπάρχει και το αντικείμενο της μητρικής κλάσης που θα δημιουργηθεί και που θα πρέπει να πάρει τιμές στις μεταβλητές μέλη του, αν θέλουμε να λειτουργήσει σωστά. Οι τρεις πρώτες παράμετροι λοιπόν, απευθύνονται προς τη μητρική κλάση και όπως βλέπετε στον κώδικα του constructor χρησιμοποιείται η σύνταξη **super.variable** για να προσπελαθεί η κάθε μεταβλητή μέλος της μητρικής κλάσης και να της ανατεθεί η αντίστοιχη τιμή.
Εδώ ακριβώς είναι που ο compiler αντιδρά, και δεν αποδέχεται το συγκεκριμένο statement. Το πρόβλημα που υπάρχει εδώ είναι πως οι μεταβλητές μέλη της μητρικής κλάσης έχουν δηλωθεί με ορατότητα **private** και άρα η παράγωγη κλάση δε μπορεί να τις προσπελάσει.
Μία πιθανή λύση στο πρόβλημα, είναι να αλλάξουμε τον προσδιοριστή ορατότητας των μεταβλητών μελών της κλάσης βάσης από **private** σε **protected**. Στον κώδικα που ακολουθεί έχει γίνει μόνο η συγκεκριμένη αλλαγή.

```
package elearning;

public class Account {
   // instance variables
   protected String holder;
   protected double balance;
   protected double interest;

   // methods
   public Account(){}

   public Account(String h, double b, double i){
      holder = h;
      balance = b;
      interest = i;
   }

   public String getHolder() {
      return holder;
   }

   public void setHolder(String h) {
      holder = h;
   }

   public double getBalance() {
      return balance;
   }

   public void setBalance(double b) {
      balance = b;
   }
```

```
    public double getInterest() {
        return interest;
    }

    public void setInterest(double i) {
        interest = i;
    }
}
```

Αν τώρα δοκιμάσουμε να κάνουμε compile τον κώδικα της **StudentAccount**, θα γίνει compiled κανονικά χωρίς κανένα πρόβλημα.

```
package elearning;

public class StudentAccount extends Account {
    // instance variables
    private double overdraft;

    // methods
    StudentAccount(){}

    StudentAccount(String h, double b, double i, double o){
        super.holder = h;
        super.balance = b;
        super.interest = i;
        overdraft = o;
    }

    public double getOverdraft() {
        return overdraft;
    }

    public void setOverdraft(double o) {
        overdraft = o;
    }
}
```

Η συγκεκριμένη λύση χρησιμοποιείται αρκετά και μάλιστα είναι πολύ πιθανό να τη δείτε και σε βιβλία, παρόλα αυτά δεν έρχεται σε συμφωνία με τους κανόνες σωστής πρακτικής και παράλληλα μπορεί να προκαλέσει προβλήματα για τα οποία θα μιλήσουμε στη συνέχεια. Στα κάποια πλεονεκτήματα της συγκεκριμένης λύσης συγκαταλλέγεται η άμεση πρόσβαση στις μεταβλητές μέλη της μητρικής κλάσης με μικρή βελτίωση της ταχύτητας μιας και δεν υπάρχει η κλήση setter μεθόδων.
Στον αντίποδα, υπάρχουν βασικά μειονεκτήματα που δημιουργούν ποικίλα προβλήματα στον κώδικά μας. Ένα από αυτά είναι πως η συγκεκριμένη λύση δεν προσφέρεται για έλεγχο εγκυρότητας τιμών και άρα η παράγωγη κλάση μπορεί να αποθηκεύσει στις μεταβλητές μέλη μη έγκυρες τιμές. Το πιο σημαντικό πρόβλημα όμως είναι η δημιουργία σχέσεων εξάρτησης μεταξύ των δύο κλάσεων. Αν χρησιμοποιήσουμε τη συγκεκριμένη λύση και αλλάξουμε κάτι στην κλάση βάσης, π.χ. αλλάξουμε το όνομα της μεταβλητής μέλους, θα πρέπει να πάμε και στην παράγωγη κλάση να πραγματοποιήσουμε την αντίστοιχη αλλαγή. Το συγκεκριμένο παράδειγμα βέβαια είναι απλό και το πρόβλημα λύνεται σχετικά εύκολα, όμως υπάρχουν περιπτώσεις που οι αλλαγές στον κώδικα της μητρικής κλάσης απαιτούν μεγάλες αλλαγές και στον κώδικα της παράγωγης.
Οι σχέσεις εξάρτησης μεταξύ των συστατικών λογισμικού θεωρούνται μεγάλο μειονέκτημα και θα πρέπει να αποφεύγονται μιας και έχουν ως αποτέλεσμα τη δημιουργία του λεγόμενου εύθραυστου λογισμικού. Για τους λόγους που αναφέραμε λοιπόν η συγκεκριμένη λύση αν και θα λειτουργήσει,

δεν σας προτείνουμε να την υιοθετήσετε μιας και αντιβαίνει στους κανόνες σωστής πρακτικής και σωστού σχεδιασμού συστημάτων.
Ακολουθεί ο κώδικας της σωστής λύσης, τόσο της κλάσης **`Account`** όσο και της **`StudentAccount`**:

```
package elearning;

public class Account {
   // instance variables
   private String holder;
   private double balance;
   private double interest;

   // methods
   public Account(){}

   public Account(String h, double b, double i){
      holder = h;
      balance = b;
      interest = i;
   }

   public String getHolder() {
      return holder;
   }

   public void setHolder(String h) {
      holder = h;
   }

   public double getBalance() {
      return balance;
   }

   public void setBalance(double b) {
      balance = b;
   }

   public double getInterest() {
      return interest;
   }

   public void setInterest(double i) {
      interest = i;
   }
}
```

```
package elearning;

public class StudentAccount extends Account {
   // instance variables
   protected double overdraft;

   // methods
   StudentAccount(){}

   StudentAccount(String h, double b, double i, double o){
      super(h, b, i);
      overdraft = o;
   }
```

```
    public double getOverdraft() {
        return overdraft;
    }

    public void setOverdraft(double o) {
        overdraft = o;
    }
}
```

Στην κλάση **Account** έχουμε επαναφέρει την ορατότητα των μεταβλητών μελών σε **private** όπως είχαμε στην αρχική έκδοση, που είναι και το σωστότερο. Η σημαντική αλλαγή που θα πρέπει να προσέξετε είναι στην κλάση **StudentAccount**, στην πρώτη γραμμή του constructor που λαμβάνει παραμέτρους. Η γραμμή αυτή είναι μία κλήση στον constructor της **Account** που αρχικοποιεί το αντικείμενο που δημιουργείται με τις τιμές των παραμέτρων. Έτσι λοιπόν, όταν δημιουργούμε ένα αντικείμενο τύπου **StudentAccount** χρησιμοποιώντας τον constructor με τις τέσσερις παραμέτρους, καλείται ο constructor της **Account** που λαμβάνει παραμέτρους για να αρχικοποιηθεί σωστά το αντικείμενο τύπου **Account** και ο constructor της **StudentAccount** απλά αρχικοποιεί τη μοναδική μεταβητή μέλος της δικής του κλάσης, που είναι μόνο η **overdraft**.
Η λύση αυτή συνάδει με τις αρχές του αντικειμενοστρεφούς προγραμματισμού, αφού κάνοντας τις μεταβλητές μέλη **private** υλοποιούμε την αρχή της ενθυλάκωσης. Επιπλέον, κάθε κλάση διαχειρίζεται αποκλειστικά και μόνο τις δικές της μεταβλητές μέλη, χωρίς να "πειράζει" μεταβλητές μέλη κάποιας άλλης κλάσης, έστω κι αν αυτή είναι η μητρική της. Με τον τρόπο αυτόν αποφεύγονται οι σχέσεις εξάρτησης μεταξύ των κλάσεων και επομένως δεν κινδυνεύουμε να καταλήξουμε με εύθραυστο λογισμικό.

5.6 Η Κλάση `java.lang.Object`

Στην υποενότητα αυτή θα μιλήσουμε για την κλάση **java.lang.Object** την οποία αναφέραμε νωρίτερα. Αναλύοντας την αρχή της κληρονομικότητας είδαμε πως η κλάση **Object** είναι η υπερκλάση όλων των κλάσεων της Java, δεδομένου πως ο compiler την ορίζει αυτόματα ως υπερκλάση σε οποιαδήποτε κλάση δεν κληρονομεί άμεσα από κάποια άλλη. Ως συνέπεια λοιπόν, όλες οι κλάσεις κληρονομούν τα χαρακτηριστικά της. Ένα από αυτά τα χαρακτηριστικά που αναφέρθηκε ήδη είναι η μέθοδος **finalize()**, την οποία μπορούν οι υποκλάσεις να υπερκαλύψουν αν αυτό είναι απαραίτητο. Εκτός από τη **finalize()**, η **Object** διαθέτει μια σειρά από μεθόδους που οι κλάσεις μας κληρονομούν από αυτήν με default συμπεριφορά, την οποία μπορούν να τροποποιήσουν υπερκαλύπτοντάς τις. Οι μέθοδοι αυτές είναι οι εξής:

1. **clone()**: δημιουργεί και επιστρέφει ένα αντίγραφο του αντικειμένου. Για να λειτουργήσει σωστά για την κλάση που δημιουργούμε, θα πρέπει να την υπερκαλύψουμε και να την υλοποιήσουμε σωστά.
2. **equals(Object)**: εξετάζει την ισότητα 2 αντικειμένων. Επιστρέφει **true** αν τα αντικείμενα είναι ίδια, αλλιώς **false**. Ομοίως θα πρέπει να υπερκαλυφθεί και να υλοποιηθεί.
3. **getClass()**: Επιστρέφει το αντικείμενο που αντιστοιχεί στην συγκεκριμένη κλάση κατά το runtime από την JVM.
4. **hashCode()**: Επιστρέφει τον κωδικό hash της κλάσης.

5. **`notify()`**: Ενημερώνει συγκεκριμένο νήμα για κάποια αλλαγή κατάστασης.
6. **`notifyAll()`**: Ενημερώνει τα νήματα σε αναμονή για κάποια αλλαγή κατάστασης.
7. **`toString()`**: Επιστρέφει ένα string σχετικό με την κλάση. Θα πρέπει να υπερκαλυφθεί.
8. **`wait()`**: Θέτει το τρέχον νήμα σε αναμονή.

5.7 Υπερκάλυψη Μεθόδων (Method Overriding)

Ο όρος «υπερκάλυψη μεθόδου» έχει ήδη αναφερθεί σε προηγούμενες ενότητες. Πρόκειται για τη διαδικασία όπου μία κλάση επανα-υλοποιεί μία μέθοδο που κληρονόμησε από κάποια μητρική της κλάση. Πρόκειται για μία πολύ κοινή περίπτωση κατά την οποία μία συγκεκριμένη συμπεριφορά υλοποιείται με διαφορετικό τρόπο στην παράγωγη κλάση από ότι στη μητρική.
Χαρακτηριστικά παραδείγματα υπερκαλυπτόμενων μεθόδων είναι αυτές της **`Object`**, τις οποίες αναφέραμε στην προηγούμενη υποενότητα. Η δυνατότητα υπερκάλυψης είναι επίσης ιδιαίτερα χρήσιμη μιας και πρόκειται για ένα από τα συστατικά που απαιτούνται για την επίτευξη πολυμορφικής συμπεριφοράς, όπως θα δούμε στη συνέχεια.
Για να υπερκαλύψουμε σωστά μία μέθοδο, θα πρέπει να προσέξουμε να μην παραβιάσουμε κάποιον από τους παρακάτω κανόνες:

- Δε θα πρέπει να αλλάξουμε τον τύπο επιστροφής της μεθόδου ή την υπογραφή της
- Μπορούμε να ελαττώσουμε ή να διαγράψουμε δηλωμένα exceptions, όχι όμως να προσθέσουμε
- Μπορούμε να δώσουμε πιο ευρεία πρόσβαση στη μέθοδο (π.χ. από **`protected`** σε **`public`**)

Έχοντας ακολουθήσει τους παραπάνω κανόνες, το μόνο που απομένει είναι γράψουμε στο σώμα της μεθόδου τον κώδικα που υλοποιεί τη νέα συμπεριφορά. Αν καλέσουμε τη μέθοδο μέσω ενός αντικειμένου της παράγωγης κλάσης θα κληθεί η μέθοδος της παράγωγης κλάσης με την νέα συμπεριφορά, ενώ αν κληθεί μέσω ενός αντικειμένου της κλάσης βάσης, θα κληθεί η δική της μέθοδος με την αρχική συμπεριφορά. Όλα αυτά θα σας γίνουν περισσότερο ξεκάθαρα, όταν αναλύσουμε την αρχή του πολυμορφισμού.

Σημείωση: Παρατηρείται πολλές φορές το φαινόμενο κάποιοι νεοεισερχόμενοι στο χώρο του αντικειμενοστρεφούς προγραμματισμούς να συγχέουν τις έννοιες της υπερφόρτωσης (overload) και της υπερκάλυψης (override) μεθόδων. Είναι πολύ σημαντικό να σας είναι ξεκάθαρο τι ακριβώς κάνει η μία και τι η άλλη.

5.8 Τελικές Κλάσεις

Φτάνοντας στο τέλος της συζήτησής μας για την αρχή της κληρονομικότητας, θα καλύψουμε το θέμα των τελικών μεθόδων και των τελικών κλάσεων. Υπάρχουν αρκετές περιπτώσεις κατά τις οποίες δεν επιθυμούμε ο προγραμματιστής να έχει τη δυνατότητα να επεκτείνει μέσω της κληρονομικότητας κάποιες από τις κλάσεις μας. Η Java μας δίνει τη δυνατότητα να αποτρέψουμε κάτι τέτοιο, κάνοντας

χρήση της δεσμευμένης λέξης **final** την οποία μέχρι στιγμής γνωρίζαμε να χρησιμοποιείται για τη δημιουργία σταθερών.
Πράγματι, η λέξη **final** εκτός από τη δημιουργία σταθερών, έχει δύο ακόμη χρήσεις:

- Όταν τοποθετείται στον ορισμό μιας κλάσης, μετατρέπει την κλάση σε τελική (final class)
- Όταν τοποθετείται στον ορισμό μιας μεθόδου, μετατρέπει τη μέθοδο σε τελική (final method)

Ορίζοντας μία κλάση ως τελική σημαίνει πως δε μπορεί να επεκταθεί μέσω κληρονομικότητας με κανέναν τρόπο. Σε οποιαδήποτε προσπάθεια δημιουργίας υποκλάσης της, ο compiler θα παράξει σφάλμα.
Από την άλλη πλευρά, κάνοντας μια μέθοδο τελική, σημαίνει πως δεν είναι δυνατόν η συγκεκριμένη μέθοδος να υπερκαλυφθεί σε κάποια υποκλάση. Και σε αυτήν την περίπτωση, αν ο προγραμματιστής προσπαθήσει να υπερκαλύψει μία τελική μέθοδο, ο compiler θα παράξει σφάλμα και η μεταγλώττιση θα αποτύχει. Η ίδια η Java περιέχει πληθώρα τελικών κλάσεων όπως για παράδειγμα η γνωστή σε όλους σας **String** αλλά και πολλές άλλες.
Θα πρέπει να τονίσουμε σε αυτό το σημείο πως οι έννοιες της τελικής κλάσης και της τελικής μεθόδου είναι διαφορετικές και δε συνδέονται μεταξύ τους με κάποιον τρόπο, είναι δηλαδή δυνατό να έχουμε μία απλή κλάση (όχι τελική) που να περιέχει τελικές μεθόδους. Οι προγραμματιστές θα μπορούν να την επεκτείνουν κανονικά μέσω κληρονομικότητας, χωρίς όμως να μπορούν να υπερκαλύψουν τις συγκεκριμένες μεθόδους.

5.9 Παραλειπόμενα

Κλείνοντας την ενότητα σχετικά με την κληρονομικότητα, θα αναφερθούμε σε κάποια παραλειπόμενα τα οποία δεν είχαμε την ευκαιρία να καλύψουμε μέχρι τώρα, ξεκινώντας από κάποιους προσδιοριστές. Εκτός από τους προσδιοριστές ορατότητας καθώς και κάποιους άλλους που έχουμε συναντήσει μέχρι στιγμής (π.χ. τον **final**) υπάρχουν και οι ακόλουθοι τέσσερις, από τους οποίους δύο χρησιμοποιούνται σε συνδυασμό με μεταβλητές μέλη και δύο με μεθόδους. Οι προσδιοριστές αυτοί είναι οι εξής:
transient: Χρησιμοποιείται με μεταβλητές μέλη μόνο. Ενημερώνει τον compiler πως δεν επιθυμούμε η συγκεκριμένη μεταβλητή μέλος να γίνει serialized ώστε να αποθηκευτεί η τιμή της στο δίσκο. Π.χ.:

```
class Experiment implements Serializable {
   transient int temperature;    // μη αποθηκεύσιμη τιμή (transient)
   double mass;                  // αποθηκεύσιμη τιμή (persistent)
   ...
}
```

volatile: Ο συγκεκριμένος προσδιοριστής χρησιμοποιείται επίσης μόνο με μεταβλητές μέλη και ενημερώνει τον compiler να μην κάνει χρήση optimizations όπως για παράδειγμα η χρήση cache. Δηλώνοντας μία μεταβλητή ως **volatile** εξασφαλίζουμε πως σε κάθε ανάγνωση ή εγγραφή της τιμής της μεταβλητής θα πραγματοποιείται διαδικασία read και write αντίστοιχα. Είναι χρήσιμος όταν έχουμε multi-threaded εφαρμογές, όπου διαφορετικά threads μπορεί να έχουν πρόσβαση σε μία μεταβλητή μέλος.

```
class SpeedMeter {
   volatile int speed;
}
```

native: Ο προσδιοριστής αυτός χρησιμοποιείται μόνο με μεθόδους των οποίων η υλοποίηση δεν είναι σε Java αλλά σε κάποια άλλη γλώσσα όπως για παράδειγμα η C++. Μία τέτοια μέθοδος μπορεί να δηλωθεί ως μέλος σε μία κλάση της Java. Δεδομένου πως η υλοποίηση της βρίσκεται σε κάποιο άλλο αρχείο, μόνο το πρότυπό της μπορεί να εμφανίζεται μέσα στο σώμα της κλάσης.
synchronized: Επίσης χρησιμοποιείται μόνο με μεθόδους, μαρκάροντάς τις ως συγχρονισμένες. Θα τις εξετάσουμε πιο αναλυτικά στην ενότητα 9.
Στον πίνακα 14 μπορείτε να δείτε συγκεντρωτικά τους προσδιοριστές που μπορούν να λάβουν τα μέλη μιας κλάσης, πλην των προσδιοριστών ορατότητας.

Προσδιοριστής	Πεδία	Μέθοδοι
static	Ορίζει μια στατική μεταβλητή	Ορίζει μια στατική μέθοδο
final	Ορίζει μια σταθερά	Ορίζει μια μέθοδο που δε μπορεί να υπερκαλυφθεί
abstract	Δε χρησιμοποιείται	Ορίζει μία αφηρημένη μέθοδο που θα υλοποιηθεί σε κάποια υποκλάση
synchronized	Δε χρησιμοποιείται	Ορίζει μία συγχρονισμένη μέθοδο
native	Δε χρησιμοποιείται	Ορίζει μία μέθοδο που έχει υλοποιηθεί σε άλλη γλώσσα
transient	Η τιμή της μεταβλητής δε θα συμπεριλφθεί κατά την αποθήκευση του αντικειμένου	Δε χρησιμοποιείται
volatile	Ο compiler δε θα χρησιμοποιήσει βελτιστοποιήσεις για την ανάγνωση και την εγγραφή τιμών για το συγκεκριμένο πεδίο	Δε χρησιμοποιείται

Πίνακας 14

Ένας βασικός τελεστής που μπορεί να χρησιμοποιηθεί για να εξετάσουμε αν ισχύει μία σχέση “IS-A” μεταξύ ενός αντικειμένου και μίας κλάσης (ή ενός interface όπως θα δούμε αργότερα), είναι ο τελεστής **instanceof**. Στην περίπτωση που η σχέση επαληθεύεται, ο τελεστής θα επιστρέψει **true**, αλλιώς θα επιστρέψει **false**. Συγκεκριμένα, ο τελεστής θα επιστρέψει **true** στις εξής περιπτώσεις:

1. Το αντικείμενο είναι στιγμιότυπο (instance) της κλάσης έναντι της οποίας γίνεται ο έλεγχος
2. Το αντικείμενο είναι στιγμιότυπο υποκλάσης της κλάσης έναντι της οποίας γίνεται ο έλεγχος
3. Το αντικείμενο είναι στιγμιότυπο κλάσης που υλοποιεί το interface έναντι στο οποίο γίνεται ο έλεγχος
4. Το αντικείμενο είναι στιγμιότυπο υποκλάσης της κλάσης που υλοποιεί το interface έναντι στο οποίο γίνεται ο έλεγχος

Για παράδειγμα, αν το **s1** είναι ένα αντικείμενο της κλάσης **MyClass**, τότε η έκφραση:

```
s1 instanceof MyClass
```

θα επιστρέψει **true**.

Ένας ακόμη χρήσιμος τελεστής, είναι ο τελεστής **this**, ο οποίος χρησιμοποιείται ως αναφορά στο τρέχον αντικείμενο. Για παράδειγμα, θα μπορούσατε για να αναφερθείτε σε μία μεταβλητή μέλος μιας κλάσης π.χ. **name** μέσα από μία μέθοδο να χρησιμοποιήσετε τη σύνταξη:

```
this.name = "Nikos";
```

Φυσικά, κάτι τέτοιο αποτελεί πλεονασμό και για τον λόγο αυτόν, το παραλείπουμε. Παρόλα αυτά, όσοι από εσάς χρησιμοποιείτε την αυτόματη παραγωγή getters/setters π.χ. του Eclipse, σίγουρα θα έχετε δει τη συγκεκριμένη σύνταξη. Ο τελεστής **this** είναι πολύ χρήσιμος όταν για παράδειγμα θέλουμε να περάσουμε το τρέχον αντικείμενο ως παράμετρο σε μία μέθοδο, π.χ.

```
doSomething(this);
```

Τέλος, θα πρέπει να αναφερθούμε και στον τελεστή εξέτασης ισότητας, όταν αυτός χρησιμοποιείται με δύο αναφορές. Έστω πως έχουμε δύο αναφορές **s1** και **s2** οι οποίες δείχνουν η κάθε μία σε ξεχωριστά αντικείμενα της ίδιας κλάσης. Υποθέστε τώρα πως και τα δύο αυτά αντικείμενα έχουν ακριβώς τις ίδιες τιμές, δηλαδή το ένα είναι αντίγραφο του άλλου.
Αν εφαρμόσετε τον τελεστή εξέτασης ισότητας στις δύο αναφορές, θα έχετε το εξής αποτέλεσμα:

```
s1 == s2    //  false!
```

Ο παραπάνω έλεγχος θα επιστρέψει **false** επειδή οι δύο αυτές αναφορές δείχνουν σε διαφορετικά αντικείμενα και επομένως περιέχουν διαφορετικές τιμές μεταξύ τους, άσχετα αν τα αντικείμενα στα οποία δείχνουν είναι πανομοιότυπα. Αν θέλατε να ελέγξετε αν τα αντικείμενα είναι ίδια, θα έπρεπε να έχετε υπερκαλύψει τη μέθοδο **equals()** της **Object** που αναφέραμε στην αντίστοιχη υποενότητα και να χρησιμοποιήσετε αυτήν για τον έλεγχο. Από τα παραπάνω είναι προφανές πως ο έλεγχος **s1 == s2** θα επιστρέψει **true** αν και οι δύο αναφορές δείχνουν στο ίδιο αντικείμενο.
Κλείνοντας την ενότητα της κληρονομικότητας, ας δούμε για ποιο λόγο είναι χρήσιμη και τι πλεονεκτήματα μας προσφέρει. Συνήθως, οι νεοεισερχόμενοι στο χώρο του αντικειμενοστρεφούς προγραμματισμού αδυνατούν να κατανοήσουν τους λόγους για τους οποίους η κληρονομικότητα είναι χρήσιμη, ενώ πολλές φορές αμφισβητούν ακόμα και το λόγο ύπαρξής της. Αυτό είναι πολύ φυσιολογικό, μιας και απαιτείται χρόνος και τριβή με το σχεδιασμό συστημάτων ώστε να φτάσει κάποιος αρχάριος στο σημείο να κατανοήσει πλήρως τα οφέλη από τη χρήση της κληρονομικότητας και να εκτιμήσει την προσφορά της στην καταπολέμηση της πολυπλοκότητας.
Η κληρονομικότητα λοιπόν είναι χρήσιμη γιατί είναι αδύνατο να προβλέψουμε εκ των προτέρων όλες τις πιθανές πληροφορίες που μπορεί να χρειαστεί να συγκρατήσει ένα συγκεκριμένο αντικείμενο. Ακόμη κι αν ήμασταν σε θέση να το κάνουμε αυτό, θα καταλήγαμε με μία κλάση «τέρας» γεμάτη μεταβλητές μέλη, που στις περισσότερες περιπτώσεις δε θα χρησιμοποιούσαμε καν. Θυμηθείτε πως το κάθε πρόβλημα είναι ξεχωριστό και χρησιμοποιεί διαφορετικό επίπεδο αφαιρετικότητας από κάποιο άλλο που μπορεί επίσης να ασχολείται με το ίδιο θέμα. Έτσι λοιπόν, αφ' ενός θα είχαμε

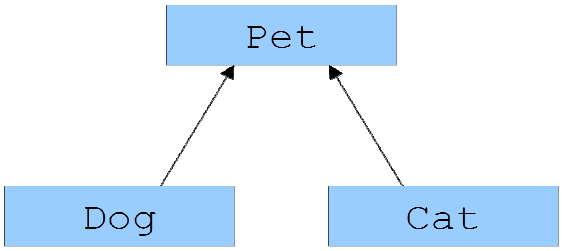
σπατάλη χώρου, αφ' ετέρου θα ήταν εξαιρετικά κουραστικό για το χρήστη να εισάγει όλον αυτόν τον όγκο πληροφοριών.
Με την κληρονομικότητα δεν έχουμε τέτοια προβλήματα αφού μια κλάση βάσης περιέχει μόνο τα απολύτως βασικά χαρακτηριστικά και συμπεριφορές του αντικειμένου που αναπαριστά. Έτσι, κάνοντας χρήση της κληρονομικότητας πετυχαίνουμε επαναχρησιμοποίηση κώδικα σε μεσαία κλίμακα. Τέλος, μέσω της κληρονομικότητας υλοποιούμε τον πολυμορφισμό, που είναι το αμέσως επόμενο θέμα που θα εξετάσουμε.

5.10 Πολυμορφισμός (Polymorphism)

Η τελευταία και μια από τις πιο σημαντικές αρχές του αντικειμενοστρεφούς προγραμματισμού που θα εξετάσουμε, είναι ο πολυμορφισμός. Στον πραγματικό κόσμο, αναφερόμαστε σε κάποιο αντικείμενο λέγοντας πως αυτό είναι πολυμορφικό, όταν έχει την ικανότητα να έχει πολλές διαφορετικές μορφές. Στον αντικειμενοστεφή προγραμματισμό ο πολυμορφισμός αναφέρεται στη δυνατότητα να χειριζόμαστε αντικείμενα που ανήκουν στην ίδια ιεραρχία κλάσεων, σαν να ήταν αντικείμενα της κλάσης βάσης. Η συγκεκριμένη δυνατότητα είναι εξαιρετικά χρήσιμη μιας και με τον τρόπο αυτόν χειριζόμαστε ομοιόμορφα τα αντικείμενα όλων των κλάσεων μιας ιεραρχίας.
Για την επίτευξη πολυμορφικής συμπεριφοράς απαιτείται μία ιεραρχία κλάσεων και υπερκαλυπτόμενες μέθοδοι.

Σημείωση: Όσοι από εσάς είστε εξοικειωμένοι με τον μηχανισμό του πολυμορφισμού από τη C++, θα γνωρίζετε σίγουρα την έννοια των εικονικών μεθόδων (virtual methods). Η Java χρησιμοποιεί εξ' ορισμού δυναμική διασύνδεση (dynamic binding) για όλες τις μεθόδους, χωρίς να απαιτείται η χρήση κάποιας ειδικής λέξης (όπως η λέξη `virtual` *στη C++).*

Για να καταλάβετε πως λειτουργεί ο πολυμορφισμός, ας δούμε το εξής παράδειγμα. Στην ιεραρχία του σχήματος 31, υπάρχει μία κλάση βάσης με το όνομα **`Pet`**, που αναπαριστά ένα κατοικίδιο ζώο. Ας υποθέσουμε πως η κλάση αυτή αποθηκεύει ως μεταβλητές μέλη το όνομα και το φύλο του κατοικιδίου. Μιας και όλα τα κατοικίδια έχουν τη δυνατότητα παραγωγής κάποιας μορφής ήχου, η κλάση θα δηλώνει και μία μέθοδο **`sound()`**.



Σχήμα 31

Δεδομένου πως η έννοια «κατοικίδιο» είναι αόριστη, η μέθοδος αυτή θα είχε μία υποτυπώδη υλοποίηση όπως η ακόλουθη:

```
public void sound(){
   System.out.println("base class call");
}
```

Από την κλάση **Pet** κληρονομούν δύο κλάσεις, η **Dog** και η **Cat** που αναπαριστούν τον σκύλο και τη γάτα αντίστοιχα. Ας υποθέσουμε για λόγους απλότητας πως καμία από αυτές δεν προσθέτει κάποια μεταβλητή μέλος ή μέθοδο. Επειδή όμως ο σκύλος παράγει διαφορετικό ήχο από αυτόν της γάτας, η μέθοδος **sound()** που κληρονομείται και από τις δύο κλάσεις, θα πρέπει να υλοποιηθεί διαφορετικά σε κάθε μία από αυτές τις κλάσεις, όπως φαίνεται στις γραμμές που ακολουθούν:

```
public void sound(){
   System.out.println("Wooof!");      // dog sound
}

public void sound(){
   System.out.println("Miaaouu!");    // cat sound
}
```

Στο κεντρικό μας πρόγραμμα, θα είχαμε τις εξής επιλογές δημιουργίας αντικειμένων και κλήσης μεθόδων:

1. **Αναφορά κλάσης βάσης σε αντικείμενο κλάσης βάσης**. Για παράδειγμα τον κώδικα:

```
Pet p = new Pet();
p.sound();
```

Στην περίπτωση αυτή που είναι και η απλούστερη, είναι προφανές πως θα κληθεί η μέθοδος της κλάσης **Pet** και άρα θα εμφανιστεί στην κονσόλα το μήνυμα:

```
Base class call
```

2. **Αναφορά παράγωγης κλάσης σε αντικείμενο παράγωγης κλάσης**, όπως φαίνεται στον κώδικα που ακολουθεί:

```
Cat c = new Cat();
c.sound();
```

Και στην περίπτωση αυτή, έχουμε ορθόδοξο χειρισμό και όπως θα περιμένατε, θα κληθεί η **sound()** της **Cat** και θα προβληθεί το μήνυμα:

```
Miaaouu!
```

3. **Αναφορά παράγωγης κλάσης σε αντικείμενο κλάσης βάσης**.

```
Cat c = new Pet();
```

Στην περίπτωση αυτή θα έχουμε σφάλμα από τον compiler, μιας και δεν ισχύει όπως έχουμε μάθει η σχέση “IS-A” αντίστροφα.

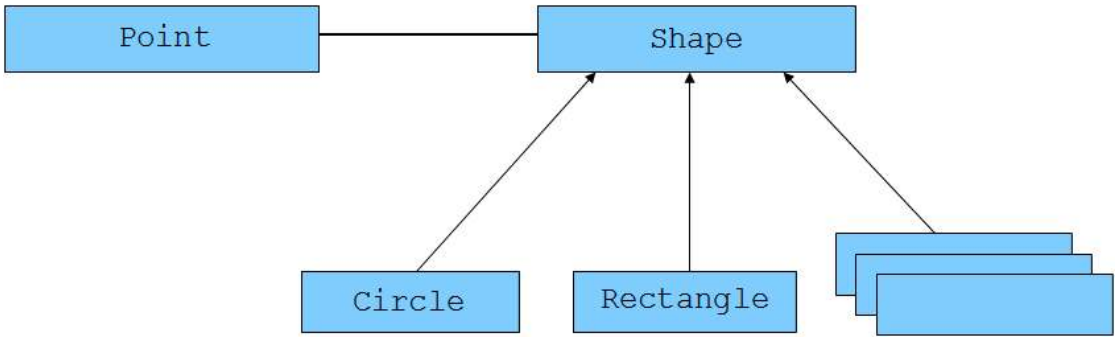
4. **Αναφορά κλάσης βάσης σε αντικείμενο παράγωγης κλάσης**: Για παράδειγμα:

```
Pet p = new Dog();
p.sound();
```

Στο σενάριο αυτό, ο κώδικας είναι απόλυτα νόμιμος μιας και ισχύει η σχέση "IS-A". Σε αυτήν την αρχή στηρίζεται και η λειτουργία του πολυμορφισμού όπου ανάλογα με το αντικείμενο στο οποίο δείχνει μια αναφορά κλάσης βάσης, η αντίστοιχη μέθοδος θα κληθεί αυτόματα χωρίς να απαιτείται κάποιος ιδιαίτερος χειρισμός από μέρους μας. Στη δεδομένη περίπτωση λοιπόν, θα κληθεί η **`sound()`** της **`Dog`** και θα εμφανιστεί στην κονσόλα το μήνυμα:

```
Wooof!
```

Έτσι λοιπόν, μέσω του πολυμορφισμού μπορούμε χρησιμοποιώντας μία αναφορά της κλάσης βάσης να καλούμε μεθόδους διαφορετικών αντικειμένων απλά εναλλάσσοντας το αντικείμενο στο οποίο δείχνει η αναφορά. Το ποια μέθοδος θα κληθεί αποφασίζεται στο runtime, ανάλογα με το αντικείμενο στο οποίο δείχνει η αναφορά.
Η συγκεκριμένη δυνατότητα είναι ένα από τα πιο ισχυρά χαρακτηριστικά του αντικειμενοστρεφούς προγραμματισμού μιας και μας επιτρέπει να γράφουμε αποδοτικό κώδικα σε υψηλό επίπεδο χωρίς να είναι απαραίτητη η χρήση δόμων επιλογής και casting. Τα συγκεκριμένα πλεονεκτήματα μπορεί προς το παρόν να μην σας είναι ορατά, είναι βέβαιο όμως πως όσοι από εσάς ασχοληθείτε με τη συγγραφή πολύπλοκων εφαρμογών θα τα εκτιμήσετε ιδιαίτερα βλέποντάς τα στην πράξη. Δεν είναι τυχαίο άλλωστε πως οι πιο προχωρημένες τεχνικές σχεδίασης (π.χ. πρότυπα σχεδίασης) χρησιμοποιούν αυτήν ακριβώς τη δυνατότητα.
Για να κατανοήσετε καλύτερα τις αρχές της κληρονομικότητας και του πολυμορφισμού, θα επιστρέψουμε στο παράδειγμα με τα γεωμετρικά σχήματα που χρησιμοποιήσαμε στην προηγούμενη ενότητα. Η λύση που θα παρουσιάσουμε στις υποενότητες αυτές θα είναι σαφώς ανώτερη αυτής της προηγούμενης ενότητας αφού θα εφαρμόσουμε τις αρχές της κληρονομικότητας και του πολυμορφισμού, παράγοντας ένα σαφώς καλύτερο σχέδιο.
Στο UML διάγραμμα του σχήματος 32 φαίνεται η ιεραρχία των κλάσεων που θα χρησιμοποιηθεί για την υλοποίηση του προγράμματος.



Σχήμα 32

Στην κορυφή της ιεραρχίας έχει βρίσκεται η κλάση **Shape**, μία γενική κλάση που αναπαριστά κάθε δισδιάστατο γεωμετρικό σχήμα και άρα περιέχει όλα τα βασικά χαρακτηριστικά και συμπεριφορές που συναντάμε σε ένα τέτοιο. Χρησιμοποιώντας την κλάση βάσης και χτίζοντας μια απλή ιεραρχία, η λειτουργία της λύσης μας έχει αλλάξει αρκετά.
Η φιλοσοφία αναπαράστασης του κάθε γεωμετρικού σχήματος έχει παραμείνει η ίδια και δεν θα επεκταθούμε στην ανάλυσή της μιας και αυτό έγινε στην προηγούμενη ενότητα. Αυτό που έχει αλλάξει όμως είναι ο τρόπος με τον οποίο επιτυγχάνεται, μιας και πλέον ο πίνακας τύπου **Point** που είναι υπεύθυνος να αποθηκεύει τα σημεία του κάθε σχήματος έχει μεταφερθεί στην βασική κλάση. Αυτό είναι απόλυτα λογικό μιας και πρόκειται για ένα από τα κοινά χαρακτηριστικά όλων των σχημάτων (κάθε ένα από αυτά θα ορίζεται από τουλάχιστον ένα σημείο).
Αντίστοιχα, στην κλάση βάσης έχουν δηλωθεί οι μέθοδοι **area()** και **perimeter()** για τον υπολογισμό του εμβαδού και της περιμέτρου αντίστοιχα, μιας και κάθε δισδιάστατο σχήμα έχει ένα εμβαδό και μία περίμετρο.
Κατά το σχεδιασμό συστημάτων, πολλές φορές βρισκόμαστε αντιμέτωποι με περιπτώσεις όπου χρειάζεται να πάρουμε αποφάσεις οι οποίες θα έχουν θετική επίδραση στο συνολικό σχέδιο και λειτουργία της λύσης μας αλλά που ίσως κάποιες φορές δεν είναι πλήρως εναρμονισμένες με τις αντίστοιχες έννοιες του πραγματικού κόσμου. Μία τέτοια περίπτωση είναι η μέθοδος υπολογισμού της περιφέρειας ενός κύκλου. Η συγκεκριμένη μέθοδος ανήκει μόνο στον κύκλο και άρα δεν έχει θέση στην κλάση βάσης, γιατί αν τοποθετηθεί εκεί θα κληρονομηθεί από όλες τις υποκλάσεις. Κάτι τέτοιο θα ήταν λάθος, μιας και δεν ορίζεται περιφέρεια τετραγώνου ή παραλληλογράμμου.
Από την άλλη πλευρά, η κληρονομούμενη από την κλάση βάσης **perimeter()**, δεν ορίζεται για τον κύκλο. Το αντίστοιχο χαρακτηριστικό για έναν κύκλο ονομάζεται περιφέρεια.
Δεδομένου του ότι πάντοτε θέλουμε να εκμεταλλευτούμε την πολυμορφική συμπεριφορά, θα συναντήσετε πολλές περιπτώσεις σαν αυτή, όπου μία έννοια 'παραποιείται' ελαφρά με στόχο να εκμεταλλευτούμε στο έπακρο τις δυνατότητες του αντικειμενοστρεφούς σχεδίου και να μη χάσουμε σε λειτουργικότητα. Έτσι λοιπόν, στη συγκεκριμένη περίπτωση χρησιμοποιούμε τη μέθοδο **perimeter()** που κληρονομεί η κλάση **Circle** για τον υπολογισμό της περιφέρειας και καταργούμε την **circumference()** που είχαμε στην προηγούμενη ενότητα. Με τον τρόπο αυτόν, έχουμε κερδίσει στο σημείο πως μπορούμε να καλέσουμε τη συγκεκριμένη μέθοδο πολυμορφικά και έχουμε χάσει στην ακρίβεια ονομασίας της μεθόδου σε σχέση με τον χρησιμοποιούμενο όρο της γεωμετρίας.
Στη συνέχεια ακολουθεί ο κώδικας κάθε κλάσης μαζί με την ανάλυση και τις απαραίτητες επεξηγήσεις. Η κλάση **Point** δεν έχει αλλάξει καθόλου, γι αυτό ο κώδικάς της έχει παραληφθεί (μπορείτε να τον δείτε στην ενότητα 4).
Ο κώδικας της κλάσης **Shape** είναι ο ακόλουθος:

```
package elearning.geometry;

public class Shape {

   // member variables
   private Point[] points;

   // methods
   // constructor that creates an "empty"
   // shape with as many points as size
   public Shape(int size){
      points = new Point[size];
```

```
    }

    // constructor that initializes
    // a shape from a Point array
    public Shape(Point[] p){
        points = p;
    }

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

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

    // behavioural methods
    public double area(){
        return 0.0;
    }

    public double perimeter(){
        return 0.0;
    }
}
```

Η κλάση **Shape** όπως προαναφέρθηκε δηλώνει ως μεταβλητή μέλος έναν πίνακα σημείων (**Point**), που χρησιμοποιείται για την αποθήκευση των σημείων που ορίζουν το κάθε σχήμα. Δηλώνει δύο constructors, έναν που λαμβάνει ως παράμετρο έναν ακέραιο που αντιστοιχεί στον αριθμό σημείων που ορίζουν το τρέχον σχήμα (π.χ. για παραλληλόγραμμο θα περνούσαμε τον αριθμό 4) και έναν που λαμβάνει ως παράμετρο έναν πίνακα σημείων. Ο μεν πρώτος απλά θα δημιουργήσει ένα αντικείμενο και θα θέσει τη μεταβλητή μέλος να δείχνει σε έναν πίνακα μεγέθους ίσου με την παράμετρο που του δόθηκε, ο δε δεύτερος δημιουργεί ένα νέο σχήμα και θέτει τη μεταβλητή μέλος να δείχνει στον πίνακα σημείων που του περάσαμε ως παράμετρο.
Πλην των getters/setters, υπάρχουν όπως προαναφέρθηκε και οι μέθοδοι **area()** και **perimeter()** οι οποίες όπως μπορείτε να δείτε από τον κώδικα, περιέχουν μία υποτυπώδη υλοποίηση (επιστρέφουν και οι δύο 0.0).
Ακολουθεί ο κώδικας της κλάσης **Circle**:

```
package elearning.geometry;

public class Circle extends Shape {

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

    // methods
    // default constructor
    public Circle() {
        super(1);
    }

    // constructor that creates a Circle
    // from a Point and a radius
    public Circle(Point c, int r){
```

```
        super(1);
        radius = r;
        getPoints()[0] = c;
    }

    // getters/setters
    public int getRadius() { return radius; }

    public void setRadius(int r) { radius = r; }

    public void displayCircleData(){
        System.out.print("center: ");
        getPoints()[0].displayCoords();
        System.out.print("radius: " + getRadius());
    }

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

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

Η κλάση **Circle** κληρονομεί από την **Shape** και έτσι χρησιμοποιεί τον πίνακα σημείων που έχει δηλωθεί σε αυτήν για να αποθηκεύσει το μοναδικό σημείο που χρησιμοποιείται, το κέντρο. Αν ρίξετε μία ματιά στον default constructor, αυτός καλεί πάντοτε τον constructor της **Shape** περνώντας ως παράμετρο τον αριθμό 1, ώστε να δημιουργηθεί ένας πίνακας ενός σημείου.
Αυτό είναι απόλυτα νόμιμο αν και λίγο ανορθόδοξο. Παρόλα αυτά, εξυπηρετεί το σκοπό μας και είναι ακόμη περίπτωση σχεδιαστικής απόφασης που πάρθηκε ώστε να κάνει τη λύση περισσότερο ευέλικτη. Ο κύκλος δηλώνει μία έξτρα μεταβλητή μέλος για να αποθηκεύσει την ακτίνα, με το όνομα **radius**.
Ο δεύτερος constructor, λαμβάνει ως παραμέτρους ένα σημείο και έναν ακέραιο που αντιστοιχεί σε μία ακτίνα και τα χρησιμοποιεί για να αρχικοποιήσει ένα αντικείμενο τύπου κύκλου. Αρχικά θα καλέσει επίσης τον constructor της **Shape** περνώντας του τον αριθμό 1 για να δημιουργηθεί ένας πίνακας ενός σημείου. Στη συνέχεια θέτει την ακτίνα ίση με την παράμετρο που περάσαμε και τέλος θέτει ως κέντρο το σημείο που περάσαμε ως παράμετρο. Η σύνταξη της τελευταίας γραμμής, **`getPoints()[0] = c;`** ίσως σας παραξενεύει αλλά δεν είναι ιδιαίτερα δύσκολο να κατανοήσετε τι κάνει. Αν δείτε τον κώδικα της **Shape**, η **`getPoints()`** είναι μία μέθοδος που επιστρέφει έναν πίνακα σημείων. Έτσι λοιπόν, στην παραπάνω γραμμή λέμε στον compiler να προσπελάσει το πρώτο σημείο του πίνακα. Πιο εύκολα θα μπορούσε να γραφτεί ως εξής:

```
Point[] p = getPoints();
p[0] = c;
```

Εδώ δηλώνουμε μία αναφορά σε πίνακα τύπου **Point** και τη θέτουμε να δείχνει στον πίνακα που επιστρέφει η **`getPoints()`**. Στη συνέχεια μέσω της αναφοράς αυτής θέτουμε την τιμή του πρώτου στοιχείου του πίνακα ίση με **c**. Το αποτέλεσμα και στις δύο περιπτώσεις είναι το ίδιο ακριβώς, απλά

στη δεύτερη δηλώνουμε μία μεταβλητή παραπάνω. Παρόλα αυτά, η δεύτερη σύνταξη ίσως σας είναι πιο εύκολη.
Τέλος, η **Circle** υπερκαλύπτει τις μεθόδους **area()** και **perimeter()** που κληρονομεί από τη **Shape** χρησιμοποιώντας τους τύπους υπολογισμού του εμβαδού και της περιφέρειας ενός κύκλου αντίστοιχα.
Ο ακόλουθος κώδικας ανήκει στην κλάση **Rectangle** που αναπαριστά ένα παραλληλόγραμμο στο επίπεδο.

```
package elearning.geometry;

public class Rectangle extends Shape {

   // default constructor
   public Rectangle() {
      super(4);
   }

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

   // constructor that creates a Rectangle
   // from another Rectangle (copy)
   public Rectangle(Rectangle r){
      super(4);
      setPoints(r.getPoints());
   }

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

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

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

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

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

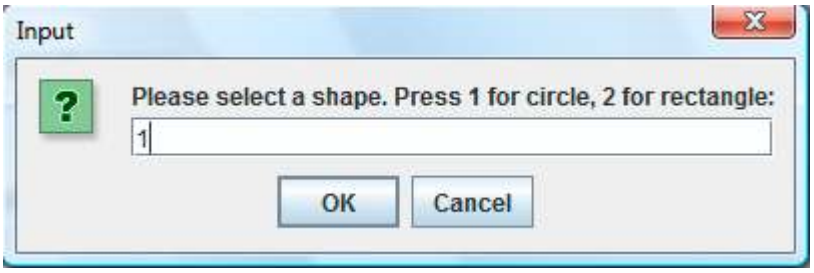
Η **Rectangle** κληρονομεί επίσης από τη **Shape** και δεν προσθέτει καμία δική της μεταβλητή μέλος. Οι μέθοδοι **area()** και **perimeter()**, όπως στην **Circle** έτσι και στη **Rectangle** υπερκαλύπτονται χρησιμοποιώντας τους τύπους υπολογισμού του εμβαδού και της περιμέτρου του παραλληλογράμμου αντίστοιχα. Η κλάση διαθέτει τρεις constructors. Ο default περνάει πάντοτε την τιμή 4 στην κλήση του constructor της μητρικής κλάσης, ώστε να δημιουργηθεί πίνακας 4 στοιχείων. Ο δεύτερος δημιουργεί ένα ολοκληρωμένο παραλληλόγραμμο χρησιμοποιώντας έναν πίνακα σημείων που δέχεται ως παράμετρο, ενώ ο τρίτος δημιουργεί ένα παραλληλόγραμμο από ένα άλλο (αντίγραφο).
Τέλος, ακολουθεί ο κώδικας της **Main** που έχει επίσης τροποποιηθεί ώστε να γίνει χρήση του πολυμορφισμού.

```
package elearning.geometry;

import javax.swing.JOptionPane;

public class Main {

   public static void main(String[] args) {

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

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

      // prompt user to make a selection
      int selection = Integer.parseInt(JOptionPane.showInputDialog(
                        "Please select a shape. Press 1 for " +
                        "circle, 2 for rectangle:"));

      // declare base class reference
      Shape s = null;

      switch (selection) {
         case 1:
            // create a circle (center p1, radius 4)
            s = new Circle(p1, 4);
            break;
         case 2:
            // create a new regtangle from the point array
            s = new Rectangle();
            s.setPoints(p);
            break;
         default:
            System.out.println("Invalid selection");
            System.exit(0);
      }

      // calculate and display circumference
      System.out.println("shape perimeter: " + s.perimeter());

      // calculate and display area
      System.out.println("shape area: " + s.area());
      System.exit(0);
   }
}
```

Όπως στην προηγούμενη ενότητα, δημιουργούνται ένας πίνακας σημείων και ένα αντικείμενο τύπου **`Point`**, τα οποία στη συνέχεια θα χρησιμοποιήσουμε για τη δημιουργία ενός παραλληλογράμμου ή ενός κύκλου αντίστοιχα.
Το πρόγραμμα θα προβάλλει έναν διάλογο στο χρήστη όπου θα πρέπει να επιλέξει τι σχήμα θέλει να δημιουργήσει. Εισάγοντας τον αριθμό 1 θα δημιουργηθεί κύκλος, ενώ αν εισάγει το 2 θα δημιουργηθεί παραλληλόγραμμο. Σε περίπτωση που εισάγει οποιονδήποτε άλλο αριθμό εκτός από 1 ή 2, θα προβληθεί μήνυμα λάθους (αν εισάγει χαρακτήρα που δεν αντιστοιχεί σε αριθμό, το πρόγραμμα θα σκάσει).
Το κλειδί επίτευξης πολυμορφικής συμπεριφοράς είναι η αμέσως επόμενη γραμμή:

```
Shape s = null;
```

Εδώ δηλώνεται μία αναφορά τύπου **`Shape`** (δηλαδή κλάσης βάσης) και ακολουθεί μία **`switch`** που ελέγχει τι σχήμα επέλεξε να δημιουργήσει ο χρήστης. Δείτε πως και στα δύο cases η αναφορά αυτή χρησιμοποιείται για τη δημιουργία του αντικειμένου, δεδομένου πως δε μπορεί να επαληθευτούν και οι δύο περιπτώσεις ταυτόχρονα. Αυτό είναι και το μοναδικό σημείο του κώδικα που χρησιμοποιείται μία δομή επιλογής για τον διαχωρισμό των αντικειμένων, μόνο κατά τη δημιουργία τους. Τα αντικείμενα δημιουργούνται με αντίστοιχο τρόπο όπως στην προηγούμενη ενότητα, απλά εμπλέκονται περισσότεροι constructors.
Από αυτό το σημείο και μετά λαμβάνει δράση ο πολυμορφισμός. Παρατηρήστε πως οι γραμμές που ακολουθούν καλούν τις μεθόδους **`area()`** και **`perimeter()`** συναρτήσει του **`s`** χωρίς να κάνουν κανέναν έλεγχο για το αντικείμενο στο οποίο δείχνει και ο κώδικας λειτουργεί σωστά.
Εκτελώντας το πρόγραμμα και επιλέγοντας δημιουργία κύκλου όπως φαίνεται στο σχήμα 33,



Σχήμα 33

παίρνουμε την ακόλουθη έξοδο:

```
shape perimeter: 25.13272
shape area: 50.26544
```

Αν τρέξετε το πρόγραμμα ξανά και επιλέξετε παραλληλόγραμμο, θα δείτε πως το πρόγραμμα λειτουργεί σωστά και θα υπολογίσει τα χαρακτηριστικά του παραλληλογράμμου.
Το πρόγραμμα του παραδείγματος είναι μικρό και ίσως δε μπορείτε να δείτε από αυτό τις τεράστιες δυνατότητες του πολυμορφισμού. Σε ένα μεγάλο πρόγραμμα με εκατοντάδες γραμμές κώδικα και πολλά διαφορετικά αντικείμενα, θα χρειαζόταν να κάνουμε χρήση δομών επιλογής κάθε φορά που θα έπρεπε να κάνουμε διαχωρισμό μεταξύ των αντικειμένων, ή να κάνουμε casting.

Με τη χρήση του πολυμορφισμού εξαλείφεται η ανάγκη αυτή και ο κώδικας λειτουργεί «δυναμικά» ελέγχοντας τι αντικείμενο υπάρχει και επιλέγοντας τι θα πρέπει να κληθεί κατά την εκτέλεση (run-time).

Σχολιάζοντας τη λύση που χρησιμοποιήσαμε σε αυτήν την ενότητα και συγκρίνοντάς την με αυτήν της προηγούμενης, είναι προφανές πως χειρίζεται σαφώς καλύτερα το πρόβλημα και μέσω των αρχών της κληρονομικότητας και του πολυμορφισμού, εχει σαφώς πιο ευέλικτη λειτουργία. Αν και για λόγους απλότητας έχουμε υλοποιήσει μόνο δύο σχήματα, είναι πολύ εύκολο να ενσωματώσουμε κώδικα για οποιοδήποτε σχήμα θελήσουμε.

Θα μπορούσαμε για παράδειγμα πολύ εύκολα να ενσωματώσουμε μία κλάση π.χ. που αναπαριστά κανονικά εξάγωνα. Το μόνο που θα χρειαζόταν θα ήταν να κληρονομήσουμε από την **`Shape`** και να υλοποιήσουμε τους αντίστοιχους constructors, ακολουθώντας την ίδια λογική για την αποθήκευση των σημείων του. Τέλος, να υπερκαλύψουμε τις **`area()`** και **`perimeter()`** χρησιμοποιώντας τους τύπους του κανονικού εξαγώνου.

Παρόλα αυτά, η λύση αυτή έχει περιθώρια βελτίωσης μιας και περιέχει και κάποιες μικρές αδυναμίες. Για παράδειγμα, στην κλάση βάσης υπάρχουν οι υποτυπώδεις υλοποιήσεις των μεθόδων **`area()`** και **`perimeter()`** που δεν κάνουν κάτι χρήσιμο. Η συγκεκριμένη αδυναμία είναι αποτέλεσμα του γεγονότος πως έχουμε μία κλάση που αναφέρεται σε μία αόριστη στον πραγματικό κόσμο έννοια, αυτήν του γεωμετρικού σχήματος.

Το γεγονός πως μπορούμε να δημιουργήσουμε στο πρόγραμμά μας αντικείμενα αυτής της κλάσης είναι κάτι που από μόνο του αποτελεί ασυμβατότητα με την αντίστοιχη έννοια στον πραγματικό κόσμο. Αν π.χ. κάποιος σας έλεγε «Ζωγραφίστε μου ένα σχήμα», τότε η πιο λογική απόκρισή σας θα ήταν, «Τι σχήμα;». Αντίστοιχα, το πρόγραμμά μας θα πρέπει να συνάδει με τις αρχές του αντικειμενοστρεφούς προγραμματισμού και να έχει αντίστοιχη λειτουργία με αυτήν του πραγματικού κόσμου. Εξαιρέσεις αποτελούν μόνο περιπτώσεις όπου γίνονται κάποιοι συμβιβασμοί αλλά προς όφελος της ευελιξίας και της λειτουργικότητας του προγράμματος (όπως π.χ. στην περίπτωση της μεθόδου υπολογισμού της περιφέρειας κύκλου).

Αν και ο τρόπος με τον οποίο χειρίζεται η συγκεκριμένη λύση το πρόβλημα δεν είναι λανθασμένος, θα δούμε στη συνέχεια πως μπορεί να βελτιστοποιηθεί κάνοντας χρήση κάποιων νέων χαρακτηριστικών που θα εξετάσουμε, εξαλείφοντας έτσι τις αδυναμίες που προαναφέρθηκαν.

5.11 Αφηρημένες Κλάσεις (Abstract Classes)

Η έννοια των αφηρημένων κλάσεων στον αντικειμενοστρεφή προγραμματισμό έρχεται να καλύψει το κενό για το οποίο μιλήσαμε στο τέλος της προηγούμενης υποενότητας, δηλαδή χρησιμοποιούνται για την αναπαράσταση αντικειμένων στον πραγματικό κόσμο που είναι αφηρημένα ως έννοιες, όπως για παράδειγμα το σχήμα, το όχημα κλπ και για τα οποία παρέχουν μία βασική υλοποίηση.

Για όσους έχουν προγραμματιστικό υπόβαθρο σε C++, οι αφηρημένες κλάσεις της Java έχουν αντίστοιχη συμπεριφορά με αυτών της C++. Το πιο ουσιώδες χαρακτηριστικό των αφηρημένων κλάσεων είναι πως δεν υπάρχει δυνατότητα δημιουργίας αντικειμένων, δηλαδή στιγμιοτύπων των συγκεκριμένων κλάσεων. Κάτι τέτοιο είναι απόλυτα λογικό, αφού με τον τρόπο αυτόν προσομοιώνεται στον κώδικα η «γενικότητα» της έννοιας που αναπαριστά στον πραγματικό κόσμο.

Αντίθετα, μπορούμε να δηλώνουμε αναφορές τύπου αφηρημένης κλάσης, κάτι που το κάνουμε πολύ συχνά για να πετύχουμε πολυμορφική συμπεριφορά.

Για να ορίσουμε μία κλάση ως αφηρημένη, χρησιμοποιούμε τη δεσμευμένη λέξη **abstract** στον ορισμό της, την οποία είχαμε συναντήσει στην ενότητα 4 μιλώντας για τις κλάσεις. Παράλληλα, θα πρέπει η κλάση να περιέχει τουλάχιστον μία αφηρημένη μέθοδο, όπως φαίνεται στο παράδειγμα του κώδικα που ακολουθεί:

```
public abstract class MyClass {
    ...
    public abstract void doThis();
}
```

Αφηρημένη ονομάζεται η μέθοδος που έχει δηλωθεί χρησιμοποιώντας τη λέξη **abstract** και επομένως δεν περιέχει υλοποίηση. Στο προηγούμενο παράδειγμα, η μέθοδος **doThis()** είναι μία αφηρημένη μέθοδος. Παρατηρήστε πως μετά τις παρενθέσεις δεν ανοίγουν άγκιστρα ώστε να υπάρξει κώδικας υλοποίησης, αλλά το statement τερματίζει με ερωτηματικό.
Έχοντας έστω και μία αφηρημένη μέθοδο στο σώμα μιας κλάσης, θα πρέπει και η ίδια η κλάση να δηλωθεί ως **abstract** αλλιώς θα προκληθεί compiler error. Το αντίστροφο δεν ισχύει, δηλαδή μπορούμε να δηλώσουμε μία κλάση ως **abstract** η οποία να μην περιέχει μία αφηρημένη μέθοδο. Αυτό βέβαια δεν έχει νόημα και φυσιολογικά δε θα πρέπει να έρθετε αντιμέτωποι με μία τέτοια περίπτωση στα σχέδιά σας.
Για ποιο λόγο όμως είναι χρήσιμο να δηλώσουμε μία μέθοδο στο σώμα μιας κλάσης και να μην την υλοποιήσουμε; Θυμηθείτε το παράδειγμα των γεωμετρικών σχημάτων και συγκεκριμένα την κλάση **Shape**, η οποία περιείχε τις μεθόδους **area()** και **perimeter()** με υποτυπώδη υλοποίηση (επέστρεφαν και οι δύο 0.0). Οι μέθοδοι αυτές ορθά βρίσκονται στο σώμα της **Shape**, μιας και σε μία κλάση βάσης τοποθετούμε τα γενικά χαρακτηριστικά των αντικειμένων που αντιπροσωπεύουν και κάθε δισδιάστατο γεωμετρικό σχήμα έχει ένα εμβαδό και μία περίμετρο. Ποιος όμως είναι ο τύπος υπολογισμού του εμβαδού και της περιμέτρου ενός "σχήματος"; Προφανώς και κάτι τέτοιο δεν υπάρχει και δεν έχει κανένα νόημα να υλοποιήσουμε τις συγκεκριμένες μεθόδους. Άρα, οι μέθοδοι **area()** και **perimeter()** αποτελούν τους ιδανικούς υποψήφιους για αφηρημένες μεθόδους.
Ως συνέπεια, και η ίδια η **Shape** θα πρέπει να μετατραπεί σε αφηρημένη που σημαίνει πως δε θα έχουμε τη δυνατότητα να δημιουργούμε αντικείμενα τύπου **Shape**. Αν όμως το εξετάσουμε και αυτό λίγο πιο προσεκτικά, θα δούμε πως δεν πρόκειται για κάποιο μειονέκτημα, μιας και πρακτικά πουθενά στον πρόγραμμά μας δε θα χρειαζόταν να δημιουργήσουμε ένα αντικείμενο τύπου **Shape**. Αντίθετα, ο συγκεκριμένος διακανονισμός κάνει το πρόγραμμά μας να αντικατοπτρίζει μια ακόμα πιο πιστή εικόνα του πραγματικού κόσμου.
Ας δούμε όμως τι συμβαίνει με τις υποκλάσεις αφηρημένων κλάσεων. Συγκεκριμένα, υπάρχουν δύο σενάρια. Το πρώτο σενάριο αναφέρεται στην περίπτωση που η άμεση υποκλάση της (αυτή που κληρονομεί κατευθείαν από την αφηρημένη) υλοποιήσει όλες τις αφηρημένες μεθόδους που κληρονόμησε. Τότε λέμε πως η συγκεκριμένη υποκλάση είναι συμπαγής. Με τον όρο συμπαγής κλάση (concrete class) αναφερόμαστε στις κλάσεις από τις οποίες μπορούμε να δημιουργήσουμε αντικείμενα. Όλες οι κλάσεις που έχουμε δει μέχρι τώρα ήταν συμπαγείς.
Στο δεύτερο σενάριο, αν παραλείψουμε να υλοποιήσουμε έστω και μία αφηρημένη μέθοδο από αυτές που κληρονομήθηκαν, η υποκλάση θα πρέπει να δηλωθεί και αυτή ως αφηρημένη, αλλιώς θα παραχθεί σφάλμα από τον compiler. Με τον τρόπο αυτόν, η ευθύνη υλοποίησης των υπολειπόμενων αφηρημένων μεθόδων μεταβιβάζεται στις κλάσεις που θα κληρονομήσουν με τη σειρά τους από την κληρονομούσα. Η πρώτη συμπαγής κλάση που θα προκύψει θα είναι αυτή στην οποία έχει υλοποιηθεί και η/οι τελευταία/ες αφηρημένη/ες μέθοδος/οι που κληρονομήθηκε/αν. Αυτό σημαίνει

πως για να έχουμε μία συμπαγή κλάση θα πρέπει στην πορεία, όλες οι αφηρημένες μέθοδοι να έχουν υλοποιηθεί σε μία ή περισσότερες υποκλάσεις της αρχικής.
Χρησιμοποιώντας στην πράξη τη θεωρία των αφηρημένων κλάσεων που καλύψαμε, το παράδειγμα των γεωμετρικών σχημάτων θα πάρει την τελική του μορφή, εξαλείφοντας παράλληλα τα μειονεκτήματα των προηγούμενων λύσεων.
Η βασική αλλαγή που έγινε ήταν να μετατραπεί η κλάση **Shape** σε αφηρημένη και ταυτόχρονα οι μέθοδοι **area()** και **perimeter()** μετατράπηκαν επίσης σε αφηρημένες και έτσι τους αφαιρέθηκε η υποτυπώδης υλοποίηση που περιείχαν.
Παράλληλα, αφαιρέθηκε η μέθοδος που υπήρχε σε κάθε σχήμα ξεχωριστά για προβολή των στοιχείων του (**displayCircleData()** και **displayRectangleData()**) και αντικαστάθηκαν από μία επίσης αφηρημένη μέθοδο που τοποθετήθηκε στην **Shape** με το όνομα **displayShapeData()**. Με τον τρόπο αυτόν, κάθε σχήμα την υπερκαλύπτει με την δική του υλοποίηση βάσει των χαρακτηριστικών του.
Η κλάση **Point** δεν τροποποιήθηκε καθόλου, ενώ στη **Main** προστέθηκε μία πολυμορφική κλήση της **displayShapeData()** ώστε να προβληθούν τα στοιχεία του σχήματος που επέλεξε ο χρήστης.
Ακολουθεί ο κώδικας των κλάσεων που τροποποιήθηκαν με τη σειρά **Shape**, **Circle**, **Rectangle** και **Main**.

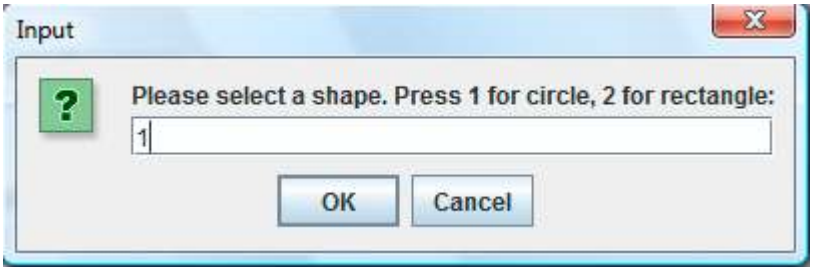
```
package elearning.geometry;

public abstract class Shape {

   // member variables
   private Point[] points;

   // methods
   // constructor that creates an "empty"
   // shape with as many points as size
   public Shape(int size){
      points = new Point[size];
   }

   // constructor that initializes
   // a shape from a Point array
   public Shape(Point[] p){
      points = p;
   }

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

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

   // behavioural methods
   public abstract double area();
   public abstract double perimeter();
   public abstract void displayShapeData();
}
```

```
package elearning.geometry;

public class Circle extends Shape {

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

   // methods
   // default constructor
   public Circle() {
      super(1);
   }

   // constructor that creates a Circle
   // from a Point and a radius
   public Circle(Point c, int r){
      super(1);
      radius = r;
      getPoints()[0] = c;
   }

   // getters/setters
   public int getRadius() { return radius; }

   public void setRadius(int r) { radius = r; }

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

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

   public void displayShapeData(){
      System.out.print("center: ");
      getPoints()[0].displayCoords();
      System.out.println("radius: " + getRadius());
   }
}

package elearning.geometry;

public class Rectangle extends Shape {

   // default constructor
   public Rectangle() {
      super(4);
   }

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

```
    // constructor that creates a Rectangle
    // from another Rectangle (copy)
    public Rectangle(Rectangle r){
        super(4);
        setPoints(r.getPoints());
    }

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

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

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

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

    public void displayShapeData(){
        System.out.println("width: " + getWidth());
        System.out.println("height: " + getHeight());
    }
}


package elearning.geometry;

import javax.swing.JOptionPane;

public class Main {

    public static void main(String[] args) {

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

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

        // prompt user to make a selection
        int selection = Integer.parseInt(JOptionPane.showInputDialog(
                            "Please select a shape. Press 1 for " +
                            "circle, 2 for rectangle:"));

        // declare base class reference
        Shape s = null;

        switch (selection) {
            case 1:
```

```
            // create a circle (center p1, radius 4)
            s = new Circle(p1, 4);
            break;
         case 2:
            // create a new regtangle from the point array
            s = new Rectangle();
            s.setPoints(p);
            break;
         default:
            System.out.println("Invalid selection");
            System.exit(0);
      }

      // display shape data
      s.displayShapeData();

      // calculate and display circumference
      System.out.println("shape perimeter: " + s.perimeter());

      // calculate and display area
      System.out.println("shape area: " + s.area());
      System.exit(0);
   }
}
```

Εκτελώντας το πρόγραμμα και επιλέγοντας τη δημιουργία κύκλου, όπως φαίνεται στο σχήμα 34,

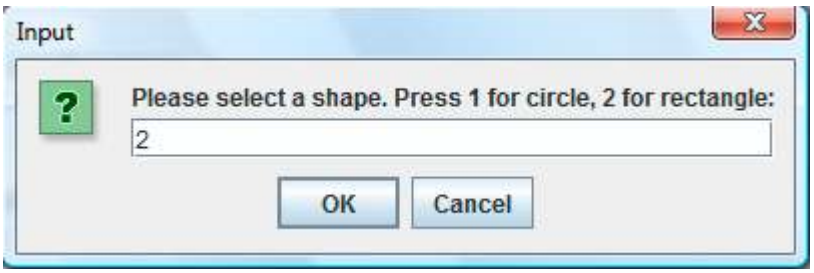


Σχήμα 34

παίρνουμε την ακόλουθη έξοδο:

```
center: x = 1, y = 1
radius: 4
shape perimeter: 25.13272
shape area: 50.26544
```

Επανεκτελώντας το πρόγραμμα και επιλέγοντας δημιουργία παραλληλογράμμου αυτή τη φορά όπως φαίνεται στο σχήμα 35,



Σχήμα 35

παίρνουμε την έξοδο που ακολουθεί:

```
width: 6
height: 3
shape perimeter: 18.0
shape area: 18.0
```

5.12 Interfaces

Όσοι από εσάς έχουν εμπειρία στον αντικειμενοστρεφή προγραμματισμό σε κάποια άλλη γλώσσα όπως π.χ. η C++, θα έχουν παρατηρήσει διαβάζοντας τα όσα έχουμε πει έως τώρα πως τα συστατικά της Java μοιάζουν πολύ με τα αντίστοιχα της C++. Για παράδειγμα και στις δύο γλώσσες υπάρχουν μεγάλες ομοιότητες στο πως υλοποιούνται οι κλάσεις, οι μέθοδοι, οι αφηρημένες κλάσεις κλπ. Εξαίρεση αποτελεί το συστατικό της γλώσσας που θα εξετάσουμε στην τρέχουσα υποενότητα, τα interfaces.

Η έννοια του interface εισήχθηκε για πρώτη φορά από την Java. Πρόκειται για ένα δομικό στοιχείο της γλώσσας που παρουσιάζει ομοιότητες με τις αφηρημένες κλάσεις, αλλά ταυτόχρονα έχει και κάποιες ουσιώδεις διαφορές. Εξ' αιτίας των ομοιοτήτων αυτών, είναι πολύ κοινό στους προγραμματιστές που βρίσκονται στο στάδιο εκμάθησης της Java να αναρωτιούνται ποια ακριβώς είναι η χρησιμότητα των interfaces.

Πριν όμως συζητήσουμε για τη χρησιμότητα των interfaces, ας δούμε κάποια από τα βασικά χαρακτηριστικά τους. Όπως οι αφηρημένες κλάσεις, έτσι και τα interfaces ορίζουν ένα νέο τύπο που όμως δε μπορεί να παράξει αντικείμενα, δηλαδή δε μπορούν στο πρόγραμμά μας να υπάρξουν στιγμιότυπα ενός interface. Τυπικά, ένα interface περιέχει ένα σύνολο από αφηρημένες μεθόδους που ορίζουν μία συμπεριφορά, την οποία συμπεριφορά αποκτούν όσες κλάσεις υλοποιήσουν το συγκεκριμένο interface υλοποιώντας τις αφηρημένες μεθόδους αυτές.

Εκτός από αφηρημένες μεθόδους, ένα interface μπορεί να περιέχει επίσης και **`static`** σταθερές. Ανεξάρτητα με το αν θα τις ορίσετε έτσι ρητά ή όχι, θα πρέπει να γνωρίζετε πως ο compiler θα προσθέσει από μόνος του τις λέξεις που λείπουν ώστε το interface που ορίζετε να περιέχει μόνο στατικές σταθερές και αφηρημένες μεθόδους.

Για να δηλώσουμε ένα interface χρησιμοποιούμε τη δεσμευμένη λέξη **`interface`**, όπως φαίνεται στο απόσπασμα που ακολουθεί.

```
public interface Doable {
   public void doThis();                    // αφηρημένη μέθοδος
```

```
    public static final int K = 5;      // στατική σταθερά
}
```

Στο παραπάνω απόσπασμα έχουμε δηλώσει ένα interface με όνομα **Doable** και ορατότητα **public**. Ένα interface μπορεί να λάβει μόνο το **public** και το *default* επίπεδο ορατότητας και γενικά, μοναδικός προσδιοριστής που μπορεί να χρησιμοποιηθεί στον ορισμό ενός interface είναι ο **public**. Μέσα στο interface έχει δηλωθεί η μέθοδος **doThis()** και η σταθερά **K**. Παρατηρήστε πως η μέθοδος δεν έχει δηλωθεί ρητά ως **abstract**, παρόλα αυτά ο compiler θα εισάγει τη λέξη που λείπει κατά τη μεταγλώττιση (δε θα φαίνεται στον κώδικα).
Για να υλοποιήσει μία κλάση ένα interface, θα πρέπει στον ορισμό της να αναφέρει ρητά το interface που υλοποιεί, όπως φαίνεται στο απόσπασμα.

```
public class MyClass implements Doable {
    ...
}
```

Στον παραπάνω κώδικα, η κλάση **MyClass** υιοθετεί τη συμπεριφορά που ορίζεται στο interface **Doable** και αναλαμβάνει την ευθύνη να υλοποιήσει τις αφηρημένες μεθόδους που ορίζονται σε αυτό, στην περίπτωσή μας την **doThis()**. Όπως στην περίπτωση των αφηρημένων κλάσεων, έτσι και εδώ αν έστω και μία από αυτές τις μεθόδους δεν υλοποιηθεί από την κλάση που κάνει implement το interface, θα πρέπει να δηλωθεί ως **abstract** αλλιώς θα έχουμε σφάλμα κατά τη μεταγλώττιση. Η ευθύνη για την υλοποίηση των μη υλοποιημένων μεθόδων μεταβιβάζεται στις υποκλάσεις της και καταλήγουμε να έχουμε συμπαγή κλάση όταν όλες οι αφηρημένες μέθοδοι του interface έχουν στην πορεία υλοποιηθεί.
Όπως μία κλάση μπορεί να επεκτείνει μία άλλη μέσω της κληρονομικότητας, έτσι και ένα interface μπορεί να επεκτείνει ένα ή περισσότερα interfaces, όπως φαίνεται στον ακόλουθο κώδικα:

```
interface BubbleBathable extends MachineWashable, Scrubable {
    ...
}
```

Το interface **BubbleBathable** κάνει χρήση της δεσμευμένης λέξης **extends** ώστε να επεκτείνει τα interfaces **MachineWashable** και **Scrubable**. Αυτό πρακτικά σημαίνει πως οποιαδήποτε κλάση κάνει implement το interface **BubbleBathable**, θα πρέπει να υλοποιήσει όλες τις μεθόδους που ορίζονται σε αυτό, συν τις μεθόδους που ορίζονται στα **MachineWashable** και **Scrubable**. Συνήθως βέβαια στα προγράμματά μας αποφεύγουμε να δημιουργούμε πολύπλοκους συσχετισμούς όπως ο παραπάνω.
Από την άλλη πλευρά, μία κλάση μπορεί να υλοποιήσει ένα ή περισσότερα interfaces:

```
public class Ball implements Bounceable, Kickable {
    ...
}
```

Η κλάση **Ball** ορίζει πως θα υλοποιήσει τα interfaces **Bounceable** και **Kickable**, που σημαίνει πως για να γίνει συμπαγής θα πρέπει να υλοποιήσει όλες τις μεθόδους που περιέχονται στο **Bounceable** καθώς και αυτές που περιέχονται στο **Kickable**. Η συγκεκριμένη δυνατότητα, ότι

δηλαδή μία κλάση μπορεί να υλοποιήσει περισσότερα το ενός interfaces προσομοιώνει με κάποιο τρόπο την πολλαπλή κληρονομικότητα της C++.

Συμβουλή για το διαγώνισμα της Oracle: Προσέξτε ερωτήσεις που προσπαθούν να σας μπερδέψουν σχετικά με τις έννοιες «υλοποιεί» (implements) και «επεκτείνει» (extends) όσον αφορά τις κλάσεις και τα interfaces. Μία κλάση μπορεί να επεκτείνει ή να επεκτεθεί από άλλες κλάσεις και να υλοποιήσει interfaces. Ένα interface μπορεί μόνο να επεκτείνει ή να επεκταθεί από άλλα interfaces.

Όπως αναφέρθηκε ήδη, ένα interface ορίζει έναν αφηρημένο τύπο δεδομένων, όπως κάνουν και οι κλάσεις. Έτσι λοιπόν, αν έχουμε στον κώδικά μας ένα αντικείμενο **b** της κλάσης **Ball** του κώδικα που προηγήθηκε, η ακόλουθη έκφραση θα επαληθευόταν:

```
b instanceof Kickable;            // true
```

Το ίδιο θα ίσχυε και για οποιαδήποτε άλλη κλάση που κληρονομεί από την **Ball**. Αυτό σημαίνει πως εκτός από ελέγχους για το αν κάποιο αντικείμενο υποστηρίζει τον τύπο που ορίζει μια κλάση, μπορούμε να ελέγξουμε και για τύπους που ορίζονται από interfaces. Τη συγκεκριμένη ιδιότητα την εκμεταλλεύονται αρκετά τόσο προγραμματιστές όσο και σύγχρονα application frameworks (π.χ. Spring) ώστε να παράγουν ευέλικτα σχέδια που δεν εξαρτώνται από συγκεκριμένη υλοποίηση, βασιζόμενα στην αρχή “Program to an interface, not a specification” (προγραμμάτισε σε ένα interface, όχι σε μία υλοποίηση). Για τον λόγο αυτόν, τα interfaces χρησιμοποιούνται πάρα πολύ στην υλοποίηση σύγχρονων εφαρμογών που κάνουν χρήση των τελευταίων τεχνολογιών.
Η πρωταρχική όμως χρήση των interfaces είναι για να ορίσουν ένα «συμβόλαιο» (contract) το οποίο θα πρέπει να τηρήσουν όλες οι κλάσεις που ενδιαφέρονται να αποκτήσουν μία συγκεκριμένη συμπεριφορά μέσω υλοποίησης κάποιου interface. Το συμβόλαιο αυτό, δρα ως «μέσο εξαναγκασμού» του προγραμματιστή να υλοποιήσει όλες τις μεθόδους του interface αν θέλει να αποκτήσει αφ’ενός τη συμπεριφορά που ορίζεται από το interface αφ’ετέρου μία συμπαγή κλάση. Στην ουσία είναι σαν να μας θέτει η γλώσσα το εξής τελεσίγραφο:
«Θες η κλάση σου να έχει την τάδε συμπεριφορά; Υλοποίησε σωστά όλες τις μεθόδους του αντίστοιχου interface». Η λέξη «σωστά» στην προηγούμενη πρόταση είναι λέξη κλειδί, μιας και η Java ναι μεν μας εξαναγκάζει να υλοποιήσουμε κάποιες μεθόδους, δεν ελέγχει όμως (και δε θα μπορούσε να το κάνει) τι ακριβώς κάνει ο κώδικας που γράφουμε. Έτσι λοιπόν, μία υλοποίηση του τύπου,

```
public void doThis() { }
```

θα κάνει compile κανονικά!
Ο μηχανισμός αυτός χρησιμοποιείται αρκετά και από την ίδια τη γλώσσα, για παράδειγμα ένας τρόπος δημιουργίας multi-threaded εφαρμογών είναι με την υλοποίηση του interface **Runnable** όπως θα δούμε στην αντίστοιχη ενότητα.
Ας δούμε πως θα μπορούσαμε να ενσωματώσουμε τη χρήση interfaces στην εφαρμογή με τα γεωμετρικά σχήματα που έχουμε χρησιμοποιήσει ως παράδειγμα. Ας υποθέσουμε πως θέλουμε να προσθέσουμε τη δυνατότητα σχεδιασμού των σχημάτων στην οθόνη του υπολογιστή (rendering) και παράλληλα αυτή η δυνατότητα να υποστηρίζεται από τα υπάρχοντα σχήματα αλλά και όποια μπορεί να προστεθούν στη συνέχεια.
Ένας τρόπος για να το πετύχουμε αυτό θα ήταν να δηλώσουμε στην **Shape** μία αφηρημένη μέθοδο π.χ. **render()** την οποία κάθε υποκλάση θα υπερκάλυπτε με την κατάλληλη ώστε να προβληθεί

σωστά στην οθόνη το κάθε σχήμα. Ένας άλλος τρόπος, θα ήταν να δημιουργήσουμε ένα interface π.χ. το **Renderable** και να θέσουμε τη **Shape** να το υλοποιεί, ως εξής:

```
public interface Renderable {
      void render();
}

public abstract class Shape implements Renderable { ... }
```

Φυσικά, η **Shape** δεν θα υλοποιούσε τη **render()** (είναι άλλωστε δηλωμένη ως **abstract**) αλλά θα μεταβίβαζε την ευθύνη υλοποίησής της στις υποκλάσεις της. Ο συγκεκριμένος τρόπος οδηγεί σε σαφώς πιο ευέλικτο σχέδιο μιας και θα μπορούσε να χειριστεί τα εξής σενάρια:

- Θέλουμε η συμπεριφορά **Renderable** να καθορίζει και τα χαρακτηριστικά προβολής των σχημάτων, π.χ. χρώματα, πάχη γραμμών κλπ. Αν τα τοποθετούσαμε όλα αυτά στην κλάση βάσης, θα έπαυε να είναι μία γενική κλάση που απλά αναπαριστά τα βασικά χαρακτηριστικά των δισδιάστατων γεωμετρικών σχημάτων.
- Αποφασίζουμε πως κάποια σχήματα δε θέλουμε (για κάποιον λόγο) να μπορούν να προβάλλονται στην οθόνη. Θα μπορούσαμε να επιστρέψουμε την **Shape** στην αρχική έκδοση (να μην υλοποιεί το **Renderable**) και να θέσουμε ξεχωριστά την κάθε κλάση σχήματος να το υλοποιεί, ανάλογα με το αν επιθυμούμε να μπορεί να γίνει rendered ή όχι.

Κλείνοντας την υποενότητα των interfaces, θα πρέπει να δώσουμε έμφαση για μία ακόμη φορά στο πόσο σημαντικά δομικά στοιχεία της γλώσσας είναι και πόσο βοηθούν στη δημιουργία ευέλικτων σχεδίων. Είναι βέβαιο πως θα τα συναντήσετε πολλές φορές, αλλά και πως θα τα χρησιμοποιήσετε και εσείς οι ίδιοι στα προγράμματά σας.

5.13 Εσωτερικές Κλάσεις (Inner Classes)

Ένας από τους βασικούς κανόνες για τη δημιουργία κλάσεων στον αντικειμενοστρεφή προγραμματισμό είναι πως αυτές θα πρέπει να περιέχουν κώδικα που περιορίζεται στον σκοπό για τον οποίο δημιουργήθηκε η κλάση. Κάθε άλλη συμπεριφορά που φανερά δεν ανήκει στη συγκεκριμένη κλάση, θα πρέπει να τοποθετείται στην κατάλληλη, για την οποία η συμπεριφορά αυτή έχει νόημα. Παρόλα αυτά, υπάρχουν περιπτώσεις κατά την υλοποίηση εφαρμογών που σχεδιάζοντας μία κλάση Α, ανακαλύπτουμε πως χρειαζόμαστε συμπεριφορά για την κλάση Α, η οποία όμως βάσει λογικής θα πρέπει να ανήκει σε ξεχωριστή κλάση Β. Το χαρακτηριστικότερο παράδειγμα ενός τέτοιου σεναρίου είναι οι event handlers (χειριστές γεγονότων) τους οποίους θα εξετάσουμε στην ενότητα 10, όταν ασχοληθούμε με την κατασκευή απλών παραθυρικών εφαρμογών.
Ο τρόπος με τον οποίο αντιμετωπίζουμε τέτοιου είδους περιπτώσεις, είναι με τη χρήση εσωτερικών κλάσεων. Όπως φαίνεται καθαρά από το όνομα, μία εσωτερική κλάση είναι μία κλάση που περιέχεται μέσα σε μία άλλη. Οι εσωτερικές κλάσεις χωρίζονται στις εξής κατηγορίες:

- Κανονικές (regular)
- Ορισμένες μέσα σε μέθοδο (method-local)
- Ανώνυμες (anonymous)
- Στατικές (static)

Θα εξετάσουμε κάθε μία από τις κατηγορίες αυτές εσωτερικών κλάσεων, ξεκινώντας από τις κανονικές. Μία κανονική εσωτερική κλάση ορίζεται εντός των αγκίστρων μιας άλλης κλάσης (την ονομάζουμε εξωτερική) όπως φαίνεται στο απόσπασμα κώδικα που ακολουθεί:

```
class Outer {
   class Inner { }
}
```

Μέχρι στιγμής, όλες οι κλάσεις που έχουμε δημιουργήσει περιέχονται η κάθε μία στο δικό της ξεχωριστό αρχείο. Αν κάνετε compile τον παραπάνω κώδικα και ελέγξετε τα αρχεία ενδιάμεσου κώδικα που παράχθηκαν, θα δείτε πως έχουν δημιουργηθεί δύο αρχεία με τα ονόματα *Outer.class* και *Outer$Inner.class*. Αυτό γίνεται γιατί ο compiler θεωρεί πως η εσωτερική κλάση είναι μία ξεχωριστή κλάση (κάτι που ισχύει) και έτσι παράγει ξεχωριστό αρχείο ενδιάμεσου κώδικα χρησιμοποιώντας τα ονόματα των δύο κλάσεων διαχωρισμένα με το σύμβολο $.
Οι εσωτερικές κλάσεις παρουσιάζουν αρκετές ιδιαιτερότητες, τις οποίες θα αναλύσουμε στην σελίδες που ακολουθούν. Για να καταλάβετε καλύτερα τη λειτουργία τους, ας δούμε το παρακάτω κομμάτι κώδικα, όπου υπάρχει μία εξωτερική κλάση με το όνομα **Outer** που περιέχει μία εσωτερική με το όνομα **Inner**.

```
package elearning;

public class Outer {
   private int x = 7;

   public static void main(String[] args){
      Outer o = new Outer();
      Outer.Inner i = o.new Inner();
      i.seeOuter();
   }

   class Inner {
      public void seeOuter(){
         System.out.println("Outer x = " + x);
      }
   }
}
```

Εκτελώντας τον παραπάνω κώδικα θα πάρουμε την ακόλουθη έξοδο:

```
Outer x = 7
```

Η **Outer** επιπλέον περιέχει μία **private** μεταβλητή μέλος τύπου **int** και όνομα **x**, στην οποία αρχικά έχει δοθεί η τιμή 7. Μία κανονική εσωτερική κλάση όπως είναι η **Inner**, όντας δηλωμένη στο σώμα μιας άλλης μπορεί να έχει οποιονδήποτε από τους προσδιοριστές ορατότητας που έχουμε μάθει για τα μέλη, δηλαδή τους **public**, **protected**, **private**, **abstract**, **strictfp**, **final** και **static**. Στο παράδειγμά μας, η **Inner** θα έχει το *default* επίπεδο ορατότητας μιας και δεν έχουμε χρησιμοποιήσει κανέναν από τους παραπάνω.
Σκοπός μιας κλάσης είναι να δημιουργεί αντικείμενα και στην περίπτωση των εσωτερικών κλάσεων αυτό μπορεί να γίνει με διαφορετικούς τρόπους από διαφορετικά σημεία του κώδικα. Η πιο απλή περίπτωση είναι όταν έχουμε δημιουργία αντικειμένου μιας inner class από κώδικα που βρίσκεται σε

μέθοδο της εξωτερικής κλάσης. Η δημιουργία αντικειμένου ακολουθεί τη σύνταξη που γνωρίζουμε, όπως φαίνεται στη γραμμή που ακολουθεί:

```
Inner i = new Inner();
```

Η δεύτερη περίπτωση καλύπτει την δημιουργία αντικειμένων είτε από στατικό κώδικα της εξωτερικής κλάσης (π.χ. κώδικα που περιέχεται σε μια static μέθοδο), είτε από κώδικα που ανήκει σε κάποια άλλη κλάση. Εδώ η σύνταξη δημιουργίας αντικειμένου διαφέρει σημαντικά από τη συμβατική και θα πρέπει να της δώσετε ιδιαίτερη προσοχή:

```
Outer.Inner i = new Outer().new Inner();
```

Αυτό που κάνει η ασυνήθιστη αυτή γραμμή είναι να δηλώσει στο αριστερό τμήμα της μία αναφορά τύπου **Inner** με το όνομα **i**. Στο τμήμα δεξιά του ίσον δημιουργεί ένα αντικείμενο τύπου **Outer** on-the-fly και αμέσως μέσω του αντικειμένου αυτού δημιουργεί ένα της κλάσης **Inner**.
Το ίδιο ακριβώς αποτέλεσμα θα μπορούσαμε να το πετύχουμε σε δύο γραμμές κώδικα ως εξής:

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
```

Αυτός είναι ο τρόπος που έχουμε χρησιμοποιήσει στο παράδειγμά μας (προσέξτε πως δημιουργούμε ένα αντικείμενο τύπου **Inner** μέσα από τη **main** που είναι πάντοτε **static**). Μέσω του αντικειμένου αυτού καλούμε τη μέθοδο της **seeOuter()** της **Inner**. Βλέποντας τον κώδικα της **seeOuter()** και την έξοδο του προγράμματος, είναι προφανές πως η εσωτερική κλάση έχει πρόσβαση σε όλες τις μεταβλητές μέλη της εξωτερικής, ανεξάρτητα από το επίπεδο ορατότητας στο οποίο έχουν δηλωθεί. Ένα ακόμη χαρακτηριστικό των εσωτερικών κλάσεων είναι πως κάθε αντικείμενο εσωτερικής κλάσης φέρει πάντοτε μία έμμεση αναφορά στην εξωτερική κλάση, γι αυτό άλλωστε μπορούμε να γράψουμε τη γραμμή,

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

και αυτή να λειτουργήσει χωρίς πρόβλημα. Τέλος, μία ακόμα ιδιαιτερότητα των κανονικών εσωτερικών κλάσεων είναι πως δε μπορούν να περιέχουν τίποτα δηλωμένο ως **static**.
Η δεύτερη κατηγορία εσωτερικών κλάσεων περιλαμβάνει αυτές που είναι δηλωμένες μέσα σε μία μέθοδο κάποιας κλάσης (method-local). Στην περίπτωση αυτή, μπορούμε να δημιουργήσουμε ένα αντικείμενο της method-local κλάσης μόνο μέσα από τον κώδικα της μεθόδου που την περικλείει και επιπλέον, ο κώδικας δημιουργίας του αντικειμένου θα πρέπει να έπεται της δήλωσης της κλάσης.
Οι method-local inner classes έχουν και αυτές άμεση πρόσβαση στις μεταβλητές μέλη της κλάσης στην οποία ανήκει η μέθοδος που τις περικλείει. Η μεγάλη ιδιαιτερότητά τους όμως έγκειται στο γεγονός πως οι method-local inner classes δεν έχουν πρόσβαση στις τοπικές μεταβλητές της μεθόδου, εκτός αν αυτές είναι δηλωμένες ως **final**.
Οι μόνοι προσδιοριστές που μπορούν να εφαρμοστούν σε κάποια method-local εσωτερική κλάση είναι ο **final** ή ο **abstract**.
Τέλος, όπως είναι λογικό, αν η μέθοδος στην οποία είναι ορισμένη η εσωτερική κλάση έχει δηλωθεί ως **static**, η εσωτερική κλάση θα έχει πρόσβαση μόνο στις στατικές μεταβλητές μέλη της εξωτερικής κλάσης.

Στο απόσπασμα κώδικα που ακολουθεί επιδεικνύεται η χρήση μιας method-local εσωτερικής κλάσης:

```
package elearning;

public class Outer2 {
   private String x = "Outer2";

   public void doStuff(){
      class Inner2 {
         public void seeOuter(){
            System.out.println(x);
         }
      }
      // instantiation of inner class
      Inner2 i = new Inner2();
      i.seeOuter();
   }

   public static void main(String[] args) {
      Outer2 ref = new Outer2();
      ref.doStuff();
   }
}
```

Εκτελώντας τον παραπάνω κώδικα, θα πάρετε την ακόλουθη έξοδο:

```
Outer2
```

Η τρίτη κατηγορία περιλαμβάνει τις ανώνυμες εσωτερικές κλάσεις. Οι κλάσεις της κατηγορίας αυτής έχουν την πιο ιδιόμορφη σύνταξη που θα συναντήσετε στη Java, προσφέρουν όμως στους προγραμματιστές μεγάλη ευκολία σε αρκετές περιπτώσεις και είναι βέβαιο πως θα τις συναντήσετε τόσο σε κώδικα όσο σε συγγράμματα, ενώ ενδεχομένως να τις χρησιμοποιήσετε και εσείς οι ίδιοι.
Οι ανώνυμες εσωτερικές κλάσεις μπορούν να οριστούν είτε μέσα σε μία κλάση, είτε μέσα στο όρισμα μιας μεθόδου κατά την κλήση της (!), που είναι και ο πιο κοινός τρόπος χρήσης τους.
Η μεγάλη χρησιμότητα των ανωνύμων κλάσεων είναι πως μας δίνουν τη δυνατότητα να δημιουργήσουμε μία κλάση on-the-fly ακριβώς τη στιγμή που τη χρειαζόμαστε. Οι κλάσεις αυτές συνήθως δεν είναι ιδιαίτερα σημαντικές αλλά είναι αναγκαίες γιατί π.χ. χρειαζόμαστε κάποιο αντικείμενο σε ένα σημείο του κώδικα που να πληροί συγκεκριμένες προϋποθέσεις και μέσω των ανωνύμων κλάσεων πληρούνται οι προϋποθέσεις αυτές με άμεσο τρόπο.
Στον κώδικα που ακολουθεί βλέπετε ένα παράδειγμα ανώνυμης κλάσης που έχει οριστεί σε μία κλάση.

```
package elearning;

public class Ball {
   public void bounce(){
      System.out.println("Ball bounces");
   }
}
```

```
package elearning;

public class BeachBall {
   Ball b = new Ball(){
```

```
        public void bounce(){
            System.out.println("anonymous ball bounces");
        }
    };

    public static void main(String[] args){
        BeachBall ref = new BeachBall();
        ref.b.bounce();
    }
}
```

Στο αρχείο *Ball.java* υπάρχει δηλωμένη μία απλή κλάση που περιέχει μόνο μία μέθοδο **bounce()**. Στο αρχείο *BeachBall.java* υπάρχει η ομώνυμη κλάση που δηλώνει ως μεταβλητή μέλος μία αναφορά τύπου **Ball**. Αυτό σημαίνει πως μπορούμε να αρχικοποιήσουμε την αναφορά αυτή να δείχνει είτε σε αντικείμενο της κλάσης **Ball**, είτε κάποιας υποκλάσης της.
Αυτό που κάνει η περίεργη σύνταξη που βλέπετε (δεξιά του ίσον) είναι να δημιουργήσει ένα αντικείμενο (με τον τελεστή **new**) μιας υποκλάσης της **Ball** που ορίζουμε on-the-fly χρησιμοποιώντας μία ανώνυμη κλάση. Η ανώνυμη αυτή κλάση κληρονομεί από την **Ball** και υπερκαλύπτει τη μέθοδο **bounce()** (δείτε πως περιλαμβάνει διαφορετικό μήνυμα). Η αναφορά τύπου **Ball** λοιπόν, αρχικοποιείται να δείχνει στο αντικείμενο αυτό.
Όταν αργότερα στην κεντρική μέθοδο δημιουργούμε ένα αντικείμενο τύπου **BeachBall** και καλούμε μέσω αυτού την **bounce()** της μεταβλητής μέλους, θα κληθεί φυσικά η μέθοδος της ανώνυμης κλάσης και θα προβληθεί το μήνυμά της. Η έξοδος που θα πάρουμε είναι η:

```
anonymous ball bounces
```

Με αντίστοιχο τρόπο μπορούμε να δημιουργήσουμε μία ανώνυμη κλάση όταν χρειαζόμαστε ένα αντικείμενο κλάσης που υλοποιεί κάποιο συγκεκριμένο interface.
Η πιο κοινή όμως μορφή χρήσης ανωνύμων κλάσεων είναι ο ορισμός τους μέσα σε ορίσματα μεθόδων (μέσα στις παρενθέσεις). Για να καταλάβετε πως ακριβώς λειτουργούν, ας υποθέσουμε πως σε ένα project έχουμε τις παρακάτω κλάσεις/interfaces.

```
package elearning;

public class Garage {
    public void parkBike(Vehicle b){
        System.out.println("bike is parked!");
    }
}
```

```
package elearning;

public interface Vehicle {
    void someMethod();
}
```

```
package elearning;

public class InnerAsArgument {

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

```
        Garage g = new Garage();
        // anonymous class as argument
        g.parkBike();
    }
}
```

Η κλάση **Garage** δηλώνει μία μέθοδο **parkBike()** η οποία δέχεται ως παράμετρο ένα αντικείμενο τύπου **Vehicle**. Το interface **Vehicle** απλά δηλώνει μία μέθοδο **someMethod()**. Η κλάση **InnerAsArgument** περιέχει την κεντρική μας μέθοδο. Εκεί δημιουργείται ένα αντικείμενο τύπου **Garage**, και θέλουμε μέσω του αντικειμένου να καλέσουμε τη μέθοδο **parkBike()**. Το πρόβλημα που υπάρχει όμως είναι πως για να κληθεί σωστά η μέθοδος χρειαζόμαστε ένα αντικείμενο κάποιας κλάσης που υλοποιεί το interface **Vehicle**. Τέτοια κλάση στο πρόγραμμά μας δεν υπάρχει και άρα δε γίνεται να καλέσουμε την **parkBike()**. Γι αυτό ο compiler μας έχει χτυπήσει λάθος.
Το πρόβλημα αντιμετωπίζεται με το να δημιουργήσουμε ένα αντικείμενο μιας ανώνυμης κλάσης on-the-fly, το οποίο θα περάσουμε ως παράμετρο στην **parkBike()** όπως φαίνεται στον κώδικα που ακολουθεί:

```
package elearning;

public class InnerAsArgument {

    public static void main(String[] args){
        Garage g = new Garage();
        // anonymous class as argument
        g.parkBike(new Vehicle(){
            public void someMethod(){
                System.out.println("someMethod");
            }
        });
    }
}
```

Η σύνταξη της ανώνυμης κλάσης του προηγούμενου κομματιού κώδικα είναι και η πιο ιδιόμορφη που θα συναντήσετε στην Java. Παρόλα αυτά, μας εξυπηρετεί πολύ μιας και μας δίνει τη δυνατότητα να δημιουργήσουμε κάποιο αντικείμενο κλάσης άμεσα και εκεί που το χρειαζόμαστε, και πρακτικά μας λύνει τα χέρια σε περιπτώσεις όπως αυτή που εξετάσαμε. Εκτελώντας την τροποποιημένη **InnerAsArgument** θα πάρετε την ακόλουθη έξοδο:

```
bike is parked!
```

Η τελευταία κατηγορία εσωτερικών κλάσεων που είναι και η πιο απλή, περιλαμβάνει τις στατικές. Οι στατικές εσωτερικές κλάσεις είναι σαν τις κανονικές, με τη μόνη διαφορά πως είναι δηλωμένες στο σώμα μιας κλάσης ως **static**, π.χ.

```
class Outer {
    static class Inner { }
}
```

Δεδομένου πως είναι **static**, μπορεί να προσπελαθεί χωρίς να απαιτείται η ύπαρξη ενός αντικειμένου της κλάσης που την περικλείει. Σε περίπτωση που θελήσουμε να αρχικοποιήσουμε ένα αντικείμενο της κλάσης αυτής, χρησιμοποιούμε τη σύνταξη:

```
Outer.Inner i = new Outer.Inner();
```

Παρατηρήστε πως σε σχέση με τις κανονικές εσωτερικές κλάσεις, δεν απαιτείται να χρησιμοποιήσουμε τον τελεστή **new** για δημιουργία αντικειμένου **Outer**.
Κλείνοντας την ανάλυση μας σχετικά με τις εσωτερικές κλάσεις, θα πρέπει να αναφέρουμε πως οι κλάσεις αυτές αποτελούν ένα ιδιόμορφο αλλά ισχυρό χαρακτηριστικό της γλώσσας, που πολλές φορές είναι ιδιαίτερα εξυπηρετικό για τον προγραμματιστή. Από όλες τις εσωτερικές κλάσεις που εξετάσαμε, οι πιο σημαντικές και αυτές που πιθανώς να χρησιμοποιήσετε και εσεις οι ίδιοι είναι οι ανώνυμες ως παράμετροι μεθόδων.

Συμβουλή για το διαγώνισμα της Oracle: Οι εσωτερικές κλάσεις δεν περιέχονται ρητά στην εξεταστέα ύλη του διαγωνίσματος της Oracle, χρησιμοποιούνται όμως σε κώδικα ερωτήσεων που εξετάζουν κάποιο άλλο objective και άρα θα πρέπει να τις γνωρίζετε. Επισήμως δηλαδή δεν ανήκουν, ανεπισήμως ανήκουν!