Αντικειμενοστρεφής Προγραμματισμός
Διάλεξη – 8 : ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ & ΠΟΛΥΜΟΡΦΙΣΜΟΣ

ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ (1/3)

- Στην αρχή της κληρονομικότητας βασίζεται ο σχεδιασμός και η υλοποίηση συστημάτων
- Η κληρονομικότητα χρησιμοποιείται κατά κόρον 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.

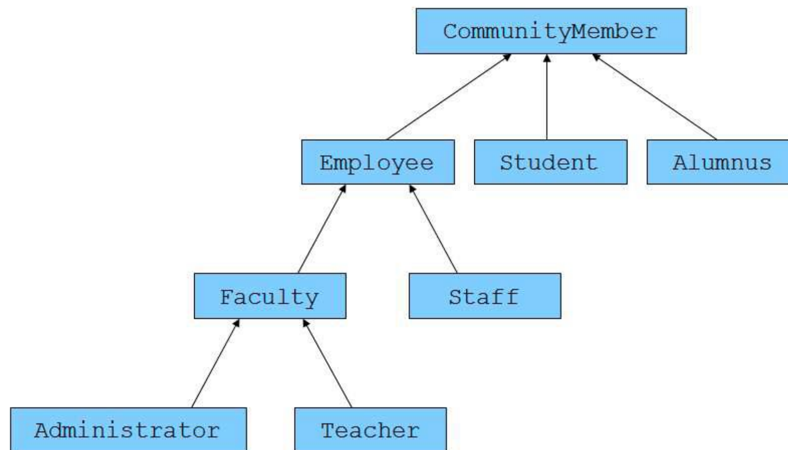
ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ (2/3)

- 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

ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ (3/3)

- Η λογική που ακολουθείται είναι να ξεκινήσουμε από μία βασική κλάση 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 συμπεριφορές.
- Έχοντας ολοκληρώσει τη διαδικασία σχεδιασμού του συστήματός μας, καταλήγουμε να έχουμε μία ιεραρχία κλάσεων (class hierarchy).
- Μία ιεραρχία κλάσεων είναι ένα "δέντρο" που διαβάζεται από επάνω προς τα κάτω. Στην κορυφή βρίσκεται η βασική κλάση, ενώ όσο προχωράμε προς τα κάτω οι κλάσεις εξειδικεύονται. Κάθε κλάση στην ιεραρχία κληρονομεί χαρακτηριστικά από αυτές που βρίσκονται επάνω από αυτήν ενώ παράλληλα κληροδοτεί σε αυτές που βρίσκονται από κάτω της.

Class Hierarchy (1/2)



Class Hierarchy (2/2)

- 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.
- 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.

Σχέσεις "IS-A", "HAS-A" (1/3)

- 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.

Σχέσεις "IS-A", "HAS-A" (2/3)

- Για να καταλάβετε καλύτερα πως λειτουργεί η σχέση "IS-A", ας δούμε το ακόλουθο παράδειγμα.
- ΠΑΡΑΔΕΙΓΜΑ
- 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**
- Η σχέση "IS-A" λοιπόν είναι άμεσα συνδεδεμένη με την αρχή της κληρονομικότητας αφού στην ουσία είναι το αποτέλεσμα μιας ιεραρχίας κλάσεων

Σχέσεις "IS-A", "HAS-A" (3/3)

- 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.
- 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.

Υλοποίηση Κληρονομικότητας (1/3)

- Η υλοποίηση της αρχής της κληρονομικότητας στον κώδικά μας γίνεται με 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 {
    ...
}
```

- Στην Java ισχύει ένας πολύ βασικός κανόνας σύμφωνα με τον οποίο η οποιαδήποτε κλάση, θα πρέπει να κληρονομεί οπωσδήποτε από κάποια άλλη. Αυτό θα σας φανεί παράξενο, αφού μέχρι στιγμής έχετε δημιουργήσει κάποιες απλές κλάσεις και ποτέ δε χρησιμοποιήσατε κληρονομικότητα.

Υλοποίηση Κληρονομικότητας (2/3)

- Όταν δημιουργούμε μία νέα κλάση, στην ουσία έχουμε δύο επιλογές. Η μία είναι να κληρονομήσουμε άμεσα από μία κλάση βάσης, όπως μάθαμε.
- 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**.

Υλοποίηση Κληρονομικότητας (3/3)

- Την κλάση **Object** θα την μελετήσουμε αργότερα. Είναι στην ουσία η υπερκλάση όλων των κλάσεων που υπάρχουν ή που μπορεί να δημιουργηθούν στη Java.
- Κανόνες που ισχύουν κατά την κληροδότηση. Αν 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";

Υλοποίηση Κληρονομικότητας (3/3)

- Την κλάση **Object** θα την μελετήσουμε αργότερα. Είναι στην ουσία η υπερκλάση όλων των κλάσεων που υπάρχουν ή που μπορεί να δημιουργηθούν στη Java.
- Κανόνες που ισχύουν κατά την κληροδότηση. Αν 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
  - 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";
```

Παράδειγμα Κληρονομικότητας (1/3)

```
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;
	}
```

Παράδειγμα Κληρονομικότητας (2/3)

```
        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;
        }
}
```

Παράδειγμα Κληρονομικότητας (3/3)

```
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;
        }
}
```

Υπερκάλυψη Μεθόδων (Method Overriding) (1/2)

- Ο όρος «υπερκάλυψη μεθόδου» αναφέρεται στη 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**.
- 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.

Υπερκάλυψη Μεθόδων (Method Overriding) (2/2)

- 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.

Τελικές Κλάσεις (Final Classes) (1/2)

- Επίσης 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. 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)

Τελικές Κλάσεις (Final Classes) (2/2)

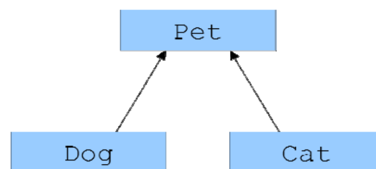
- Ορίζοντας μία κλάση ως τελική σημαίνει πως δε μπορεί να επεκταθεί μέσω κληρονομικότητας με κανέναν τρόπο. Από την άλλη πλευρά, κάνοντας μια μέθοδο τελική, σημαίνει πως δεν είναι δυνατόν η συγκεκριμένη μέθοδος να υπερκαλυφθεί σε κάποια υποκλάση.
- Η ίδια η Java περιέχει πληθώρα τελικών κλάσεων όπως για παράδειγμα η γνωστή σε όλους σας **String** αλλά και πολλές άλλες.
- 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.

Πολυμορφισμός Poymorphism (1/3)

- Ο πολυμορφισμός είναι μια από τις πιο σημαντικές αρχές του αντικειμενοστρεφούς προγραμματισμού.
- Στον πραγματικό κόσμο, αναφερόμαστε σε κάποιο αντικείμενο λέγοντας πως αυτό είναι πολυμορφικό, όταν έχει την ικανότητα να έχει 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.

Πολυμορφισμός Poymorphism (2/3)

- Για να καταλάβουμε πως λειτουργεί ο πολυμορφισμός, ας δούμε το εξής παράδειγμα όπου 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()**



- 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");
}
```

Πολυμορφισμός Poymorphism (3/3)

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
}
```

- **Παραδείγματα από calls**
- Pet p = **new** Pet();
  p.sound();
- Cat c = **new** Cat();
  c.sound();
- Pet p = **new** Dog();
  p.sound();

Παράδειγμα Πολυμορφισμού (1/5)

```
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;
        }
}
```

Παράδειγμα Πολυμορφισμού (2/5)

```
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;
}
```

Παράδειγμα Πολυμορφισμού (3/5)

```
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();
}
}
```

Παράδειγμα Πολυμορφισμού (4/5)

```
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;
```

Παράδειγμα Πολυμορφισμού (5/5)

```
			case 2:
				// create a new rectangle 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);
	}
}
```