Αντικειμενοστρεφής Προγραμματισμός
Διάλεξη – 9 : ΑΦΗΡΗΜΕΝΕΣ ΚΛΑΣΕΙΣ
INTERFACES
ΕΣΩΤΕΡΙΚΕΣ ΚΛΑΣΕΙΣ (INNER CLASSES)

Αφηρημένες Κλάσεις (Abstract Classes) (1/6)

- Οι αφηρημένες κλάσεις 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.
- 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.

Αφηρημένες Κλάσεις (Abstract Classes) (2/6)

- Για να ορίσουμε μία κλάση ως αφηρημένη, χρησιμοποιούμε τη δεσμευμένη λέξη **abstract** στον ορισμό της.
- Παράλληλα, θα πρέπει η κλάση να περιέχει τουλάχιστον μία αφηρημένη μέθοδο, όπως φαίνεται παρακάτω:

```
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 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.

Αφηρημένες Κλάσεις (Abstract Classes) (3/6)

- 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.
- 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).

Αφηρημένες Κλάσεις (Abstract Classes) (4/6)

- 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.

Αφηρημένες Κλάσεις (Abstract Classes) (5/6)

- Τι συμβαίνει με τις υποκλάσεις αφηρημένων κλάσεων? Συγκεκριμένα, υπάρχουν δύο σενάρια.
  - Η άμεση υποκλάση (αυτή που κληρονομεί κατευθείαν από την αφηρημένη) υλοποιεί 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.
  - Περίπτωση που 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.

Αφηρημένες Κλάσεις (Abstract Classes) (6/6)

- 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.

Παράδειγμα Αφηρημένων κλάσεων (1/6)

```
public abstract class Shape {
        // member variables
        private Point[] points;
        // 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();
}
```

Παράδειγμα Αφηρημένων κλάσεων (2/6)

```
public class Circle extends Shape {
        private int radius;
        public static final double PI = 3.14159;
        public Circle() {
                super(1);
        }
        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 double area() {
                return PI * radius * radius;
        }
        public double perimeter(){
                return 2 * PI * radius;
        }
        public void displayShapeData(){
                System.out.print("center:");
                getPoints()[0].displayCoords();
                System.out.println("radius:" + getRadius());
        }
}
```

Παράδειγμα Αφηρημένων κλάσεων (3/6)

```
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);
        }
        public Rectangle(Rectangle r){
                super(4);
                setPoints(r.getPoints());
        }
        public int getWidth(){
                return getPoints()[1].getX() - getPoints()[0].getX();
        }
```

Παράδειγμα Αφηρημένων κλάσεων (4/6)

```
        public int getHeight(){
                return getPoints()[1].getY() - getPoints()[2].getY();
        }
        public double area(){
                return getWidth() * getHeight();
        }
        public double perimeter(){
                return 2 * getWidth() + 2 * getHeight();
        }
        public void displayShapeData(){
                System.out.println("width: " + getWidth());
                System.out.println("height: " + getHeight());
        }
}
```

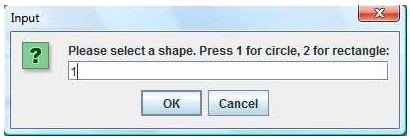
Παράδειγμα Αφηρημένων κλάσεων (5/6)

```
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:
```

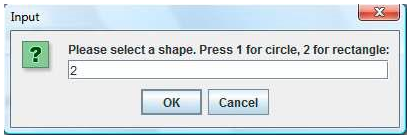
Παράδειγμα Αφηρημένων κλάσεων (6/6)

```
                    // 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);
            }
}
```

Program Run



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



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

Interfaces (1/9)

- 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.
- Πριν πούμε για τη χρησιμότητα των interfaces, ας δούμε κάποια από τα βασικά χαρακτηριστικά τους.

Interfaces (2/9) – Χαρακτηριστικά των interfaces

- 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, ο compiler θα προσθέσει από μόνος του τις λέξεις που λείπουν ώστε το interface που ορίζετε να περιέχει μόνο στατικές σταθερές και αφηρημένες μεθόδους.

Interfaces (3 / 9) – Δήλωση interfaces

- Για να δηλώσουμε ένα interface χρησιμοποιούμε τη δεσμευμένη λέξη **interface,** όπως φαίνεται στο απόσπασμα κώδικα που ακολουθεί :

```
public interface Doable {
        public void doThis(); // αφηρημένη μέθοδος
        public static final int K = 5; // στατική σταθερά
}
```

Interfaces (4 / 9) – Δήλωση interfaces

-
```
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).

Interfaces (5 / 9) – Υλοποίηση interfaces

- 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.

Interfaces (6 / 9) –Επέκταση interfaces

- 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**.

Interfaces (7 /9) –Υλοποίηση πολλαπλών interfaces

- 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**.

Interfaces (8 /9)

- 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 ώστε να παράγουν ευέλικτα σχέδια που δεν εξαρτώνται από συγκεκριμένη υλοποίηση, βασιζόμενα στην αρχή "**Program to an interface, not a specification**". Για τον λόγο αυτόν, τα interfaces χρησιμοποιούνται πάρα πολύ στην υλοποίηση σύγχρονων εφαρμογών που κάνουν χρήση των τελευταίων τεχνολογιών.

Interfaces (9 / 9)

- Η πρωταρχική χρήση των interfaces είναι για να ορίσουν ένα «συμβόλαιο» το οποίο θα πρέπει να τηρήσουν όλες οι κλάσεις που θέλουν 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 έχουμε ένα 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 μας εξαναγκάζει να υλοποιήσουμε κάποιες μεθόδους.

Εσωτερικές κλάσεις (Inner Classes) (1/2)

- 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 όταν ασχοληθούμε με την κατασκευή απλών παραθυρικών εφαρμογών.

Inner Classes (2/2)

- 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 δούμε μόνο την περίπτωση των κανονικών εσωτερικών κλάσεων μιας και οι υπόλοιπες είναι κάτι πολύ λεπτομερές και έξω από τον σκοπό του μαθήματος

Regular Inner Classes (1/)

- 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 δούμε μόνο την περίπτωση των κανονικών εσωτερικών κλάσεων μιας και οι υπόλοιπες είναι κάτι πολύ λεπτομερές και έξω από τον σκοπό του μαθήματος

Regular Inner Classes (1/2)

- 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 $.

Regular Inner Classes (2/2)

```
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);
                }
        }
}
```