



Προγραμματισμός II (Java)

2. Μέθοδοι

Περιεχόμενα

- Μέθοδοι
 - Πέρασμα αντικειμένου σε μέθοδο
 - Υπερφόρτωση μεθόδων
 - Μέθοδοι κατασκευαστές
 - Υπερφόρτωση κατασκευαστών
- Επαναχρησιμοποίηση κώδικα
 - Σύνθεση και κληρονομικότητα
 - Πολυμορφισμός
- Αφηρημένες κλάσεις
- Διεπαφές



Μέθοδοι

Δομή μεθόδου

```
πρόσβαση τύποςΕπιστροφής όνομαΜεθόδου (τύπος1 παράμετρος1, ...)  
{  
  ...  
}
```

```
public double addDouble (double num1, double num2)  
{  
  return num1+num2;  
}
```

Κλήση και αποτέλεσμα

- Όταν καλούμε μια μέθοδο αυτή
 - είτε επιστρέφει μια τιμή
π.χ. `public String getName(){...}`
 - είτε δεν επιστρέφει τίποτε
π.χ. `public void setName(String n){...}`
- Για να καλέσουμε μια μέθοδο, (συνήθως) χρειαζόμαστε ένα αντικείμενο της αντίστοιχης κλάσης
 - `Human h1=new Human();`
 - `h1.setName("John");`
- Όταν η μέθοδος επιστρέφει τιμή τότε πρέπει να την κρατάμε σε μια προσωρινή μεταβλητή. Διαφορετικά η επιστρεφόμενη τιμή χάνεται
 - `String temp = h1.getName();`

static μέθοδοι

- Καλούνται απευθείας μέσω της κλάσης ή μέσω αντικειμένων της κλάσης
- Οι static μέθοδοι έχουν πρόσβαση στα static μέλη της τάξης

Πέρασμα αντικειμένου σε μέθοδο

- Όταν περνάμε ένα αντικείμενο σε μια μέθοδο, το πέρασμα γίνεται με αναφορά στο πραγματικό αντικείμενο.

```
public void giveName(Human h, String n){  
    h.setName(n);  
}
```

- Όποιες αλλαγές γίνουν στο αντικείμενο εντός της μεθόδου, περνούν μόνιμα στο αντικείμενο

```
Human nonos=new Human();  
Human paidi=new Human();  
nonos.giveName(paidi, "Mary");
```



Υπερφόρτωση μεθόδου

Υπερφόρτωση μεθόδων

- Σε μια τάξη μπορούμε να ξαναχρησιμοποιήσουμε το ίδιο όνομα για μεθόδους που έχουν ελαφρώς διαφορετική συμπεριφορά
 - Διαφορετικούς τύπους ορισμάτων
 - Διαφορετικό πλήθος ορισμάτων

`void speak (String phrase);`

`void speak ();`

`void speak (int times, String phrase);`

Περισσότερη υπερφόρτωση

- Δεν μπορούμε να χρησιμοποιήσουμε υπερφόρτωση μόνο στην επιστρεφόμενη τιμή
`boolean speak(String phrase);` `void speak (String phrase);`
Δεν είναι φανερό ποια συνάρτηση από τις δύο θα χρησιμοποιηθεί
- Γιατί να προσθέσουμε υπερφόρτωση;
οικονομία σε ονόματα
ευανάγνωστος κώδικας
βιβλιοθήκες, λιγότερα ονόματα για να μάθει κανείς
- Μειονεκτήματα
Γίνονται λάθη ευκολότερα
μερικές φορές εμφανίζονται παραπλανητικά μηνύματα λάθους

Κλήση υπερφορτωμένης μεθόδου

- Ο compiler ψάχνει αυτήν που ταιριάζει καλύτερα
 - Προτιμά ακριβές ταιρίασμα από όλα τα άλλα
 - Βρίσκει την πιο κοντινή προσέγγιση
 - Επιδιώκει μόνο τις μετατροπές που έχουμε διεύρυνση τύπων και όχι στένεμα π.χ.
 - `void add (int l, int j);`
 - `void add (double d, double e);`
 - Η `add(10,8)` θα χρησιμοποιήσει το `(int, int)`
 - Η `add(3.5, 4)` θα χρησιμοποιήσει το `(double, double)`
- Μπορούμε να χρησιμοποιήσουμε `casting` για να επιβάλλουμε μια επιλογή, ή για να αποφύγουμε σύγχυση

Παράδειγμα

```
class Calculator{
    static int add(int a, int b) {return a+b;}
    static float add(float a, float b) {return a+b;}
    static double add(double a, double b) {return a+b;}

    public static void main(String args[] ){
        int x=5; int y=6;
        double k=5.3; double m=4.5;
        System.out.println("Atrhoisma "+ add(x,y));
        System.out.println("Athroisma "+ add(k,m));
        System.out.println("Athroisma "+ add(x,m));
    }
}
```

Μέθοδος κατασκευαστής

■ Βασικός ή κενός κατασκευαστής:

- Μια public μέθοδος με όνομα ίδιο με αυτό της τάξης, χωρίς παραμέτρους και τύπο επιστροφής.
- `public Human(){ }` – Δεσμεύει μνήμη για το αντικείμενο και κάνει τις αρχικοποιήσεις
- Αν τον ορίσουμε στην τάξη, μπορούμε να ορίσουμε τις αρχικοποιήσεις που θα κάνει
- Μπορούμε να ορίσουμε άλλους κατασκευαστές. Οπότε αναιρείται ο βασικός. Αν θέλουμε και το βασικό κατασκευαστή πρέπει υποχρεωτικά να τον ορίσουμε.

Παράδειγμα (1)

- Ορισμός ενός καλύτερου κατασκευαστή για τη Human

```
public Human (String tempName, String tempSurname,  
int tempAge)  
{  
    name=tempName;  
    surname=tempSurname;  
    age=tempAge;  
}
```
- Αν ορίσουμε μόνο αυτόν τον κατασκευαστή τότε στη demo θα μπορούμε να χρησιμοποιούμε μόνο αυτόν.
Human h1=new Human(); // βγάζει λάθος

Παράδειγμα (2)

- Ορισμός του βασικού κατασκευαστή στη Human

```
public Human(){  
    name="";  
    surname="";  
    age=0;  
}
```

- Διαφορετικά τα name και surname σε κάθε νέο Human είναι null.

```
Human h1=new Human();  
System.out.println(h1.name); //τυπώνει null
```

Η λέξη **this**

- Χρησιμοποιείται μέσα σε μια μέθοδο για να αναφερθούμε στο αντικείμενο για το οποίο καλείται η μέθοδος.
- Το **this** είναι μια αναφορά στο αντικείμενο στο οποίο βρισκόμαστε.
- Αν για παράδειγμα μια μέθοδος της τάξης Human έπρεπε να επιστρέφει το ίδιο το αντικείμενο:

```
Human increaseAge(){  
    age++;  
    return this;  
}
```

- `Human h1=new Human("Nikos", "Nikolaou",20);`
- `int x=h1.increaseAge().increaseAge().getAge();`

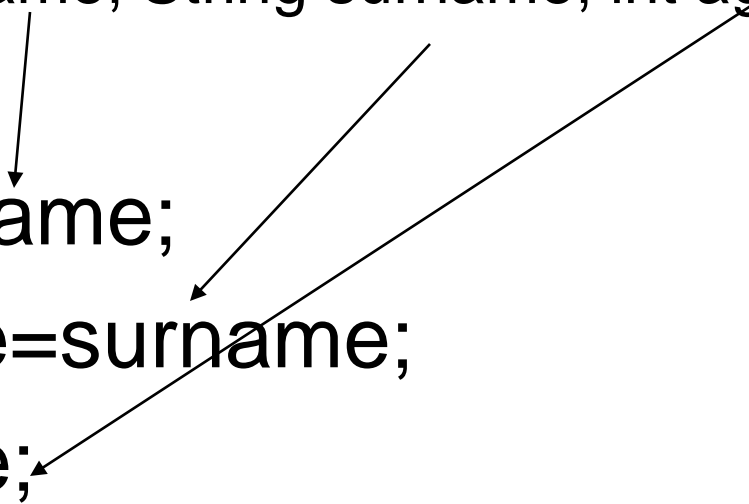
Το **this** και οι κατασκευαστές

- Με τη χρήση του **this** μπορεί να καλεί ο ένας κατασκευαστής τον άλλο.

```
public Human (String tempName, String tempSurname)
{
    this (tempName, tempSurname,0);
}
```

Το **this** και οι συνωνυμίες μεταβλητών

```
public Human (String name, String surname, int age)
{
    this.name=name;
    this.surname=surname;
    this.age=age;
}
```



Η μέθοδος finalize()

- Η Java διαθέτει μηχανισμό απελευθέρωσης της μνήμης που δεσμεύουμε και παύουμε να χρησιμοποιούμε (garbage collector)
- Ο μηχανισμός απελευθερώνει μνήμη που δεσμεύτηκε με new (π.χ. όταν βγούμε από το μπλοκ που έγινε η δέσμευση)
- Πριν ο garbage collector αποδεσμεύσει το χώρο ενός αντικειμένου καλεί τη finalize.

Πότε ορίζουμε την finalize

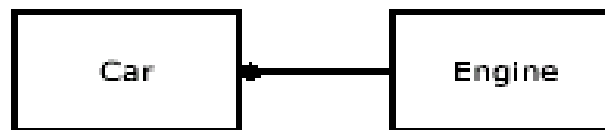
- Το garbage collection δεν σημαίνει απόλυτη διαγραφή των πράξεων του αντικειμένου,
 - π.χ. αν το αντικείμενο έχει αυξήσει μια static μεταβλητή/ μετρητή θα πρέπει να τη μειώσουμε στη finalize()
- Το garbage collection δεν γίνεται άμεσα,
 - π.χ. συμβαίνει όταν χρειαστεί μνήμη το πρόγραμμα
- Το garbage collection απλά αποδεσμεύει μνήμη
 - π.χ. αν θέλουμε να παρακολουθούμε πότε αποδεσμεύεται μνήμη μπορούμε να ορίσουμε τη finalize() να τυπώνει κάποιο μήνυμα



Επαναχρησιμοποίηση κώδικα

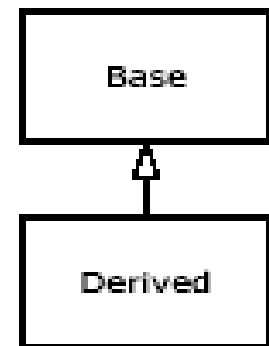
Επαναχρησιμοποίηση κλάσεων

- Δημιουργία ενός αντικειμένου μιας κλάσης
- Χρήση μιας κλάσης στον ορισμό μιας άλλης κλάσης - Σύνθεση (composition ή aggregation)
 - π.χ. Φτιάχνουμε μια κλάση «μηχανή» και στη συνέχεια μια κλάση «αυτοκίνητο» που «έχει» μία «μηχανή»



Κληρονομικότητα

- Αντιγραφή της δομής της κλάσης και επέκτασή της με νέα χαρακτηριστικά και λειτουργίες.
- Αν αλλάξει η βασική κλάση (base ή super ή parent class), τότε αλλάζει και η παράγωγη κλάση (derived ή inherited ή sub ή child class).
- Στη Java κάθε τάξη μπορεί να κληρονομεί μόνο μία κλάση



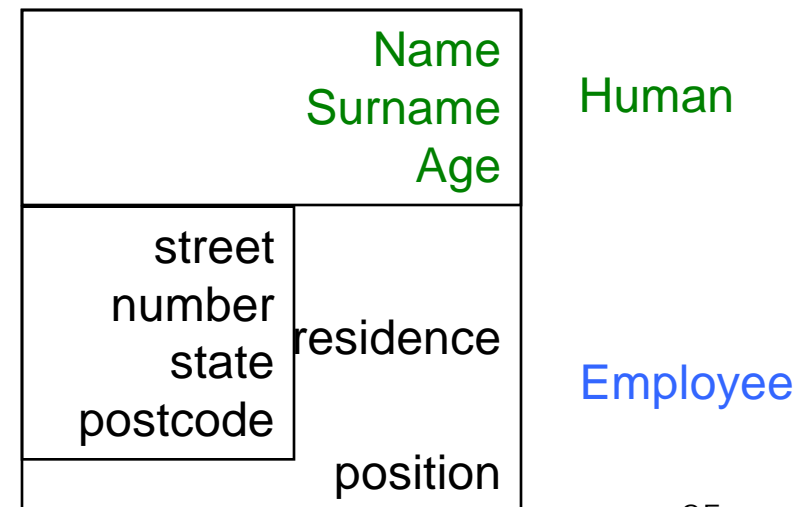
Παράδειγμα

```
class Human{
    String name;
    String surname;
    int age;
    void setName(String tempName){name=tempName;}
    String getName(){return name;}
    ...
}
class Address{
    String street;
    int number;
    String state;
    long postcode;
}
```

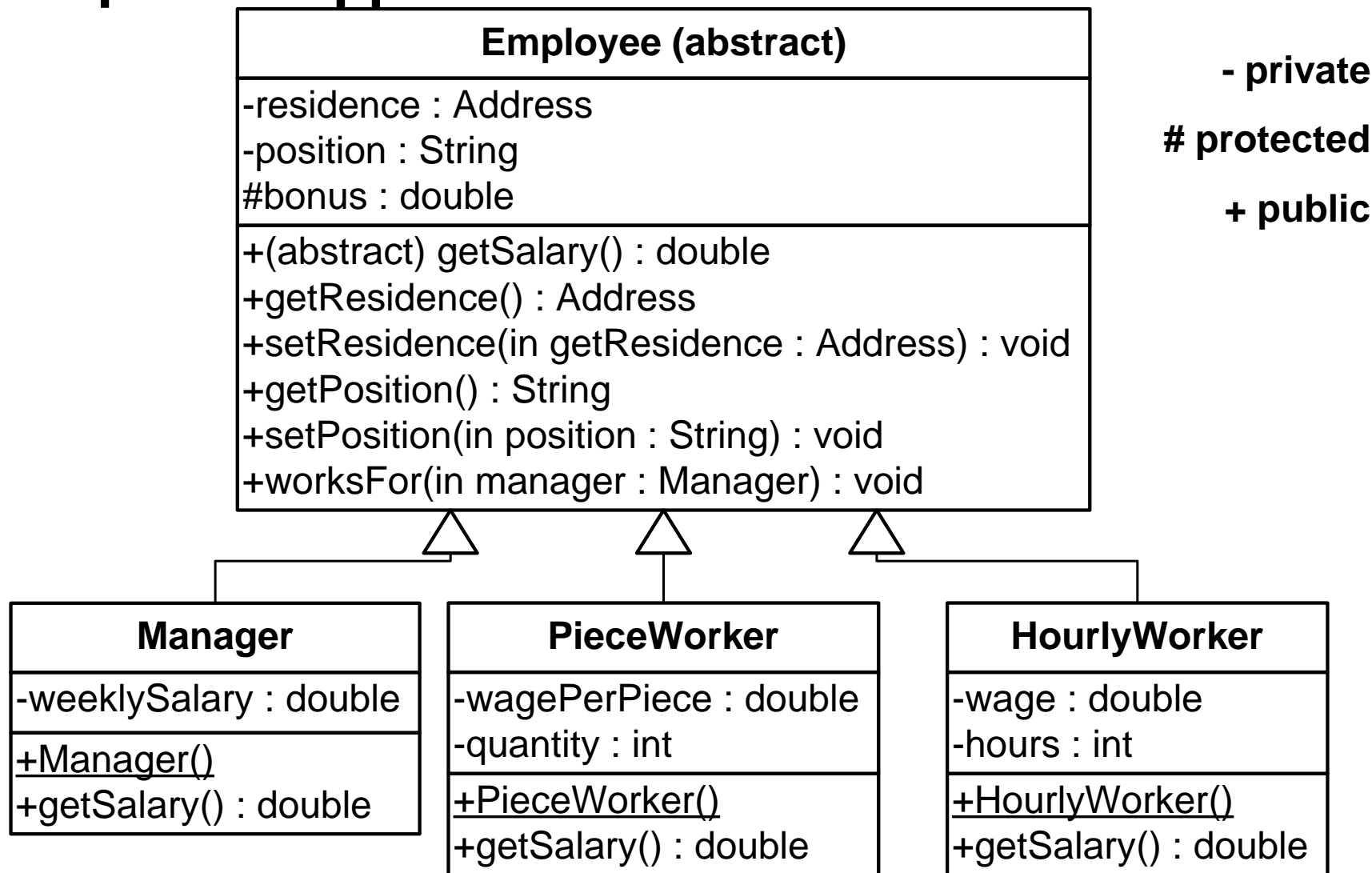

Παράδειγμα

```
public class Employee extends Human{ //κληρονομικότητα
    String position;
    Address residence; //σύνθεση
    void setPosition(String temp){position=temp;}
    String getPosition () {return position;}
    void setAddress(Address tempad){residence=tempad;}
    Address getAddress() {return residence;}
}

... main(..){
Employee e1=new Employee();
Address a1=new Address();
a1.street="Patision"; ...
e1.setAddress(a1);
}
```



Παράδειγμα



Κληρονομικότητα και ορατότητα

■ Από τα μέλη της Employee:

- Στην Employee και τα αντικείμενά της: είναι όλα ορατά private, protected και public
- Στις Manager, PieceWorker, HourlyWorker και τα αντικείμενά τους: είναι μόνο τα public και protected (π.χ. το bonus και όλες οι μέθοδοί της) π.χ. στην Manager:

```
double getSalary(){ return weeklySalary*4+bonus;}  
public Manager(){  
    position="manager"; //ΛΑΘΟΣ: το position  
    είναι protected  
    setPosition("manager"); //ΣΩΣΤΟ  
}
```

- Σε οποιαδήποτε άλλη κλάση: είναι μόνο τα public (π.χ. οι μέθοδοι)

Κληρονομικότητα και κατασκευαστές

- Οι κατασκευαστές δεν κληρονομούνται
 - Είναι στενά δεμένοι στην κλάση με τον ορισμό τους
- Αν η Employee δεν ήταν abstract και όριζε δύο κατασκευαστές

```
public Employee(String nm, int wage, int hours,  
    double attitude)
```

```
public Employee(String nm, int wage)
```

- και η Manager όριζε ένα constructor

```
public Manager(String nm, int wage, int hours, double  
    attitude, Employee under)
```

- Τότε η Manager δεν μπορεί να κληρονομήσει τους κατασκευαστές των 2 και 4 ορισμάτων

```
Manager b = new Manager("Ralph", 25); // Λάθος
```

```
Manager s = new Manager("Pat", 25, 50, .8, null); // Σωστό
```

κλάσεις αφηρημένου τύπου (abstract)

- Οι αφηρημένες κλάσεις σχεδιάζονται για να οργανώσουμε μία κοινή ιδιότητα, οι οποίοι όμως δεν χρησιμοποιείται από μόνη της
- π.χ. Τετράπλευρο, Έλλειψη, Πολύγωνο ..

- έχουν πολλά κοινά



- Θέλουμε να οργανώσουμε αυτή την κοινή συμπεριφορά σε έναν γονέα γενικού τύπου
 - με το όνομα "Shape"

Κληρονομικότητα και συμβατότητα

- Εφόσον η Employee είναι abstract δεν μπορούμε να δημιουργήσουμε αντικείμενά της. Μόνο αντικείμενα των παράγωγων κλάσεων.
- Η Manager μπορεί να αντικαταστήσει την Employee επειδή έχει όλες τις ιδιότητες της (+ άλλες)
 - Οτιδήποτε δουλεύει με την Employee δουλεύει και με την Manager επίσης
- Το αντίστροφο δεν ισχύει
 - π.χ η μέθοδος worksFor της Employee δεν μπορεί να πάρει αντικείμενο Employee σαν παράμετρο.

Δομές και κληρονομικότητα

- Σε ένα array με αντικείμενα Human μπορούμε να βάλουμε και αντικείμενα Employee (***upcasting***)
Human[] group=new Human[];
group[0]=new Human();
group[1]=new Employee();
- Στα αντικείμενα αυτά μπορούμε χωρίς κίνδυνο να καλέσουμε χαρακτηριστικά και μεθόδους της Human.
- Για να καλέσουμε χαρακτηριστικά και μεθόδους της Employee από κάποιο αντικείμενο πρέπει πρώτα να το μετατρέψουμε σε Employee (***downcasting***)
(Employee)group[1].getPosition();
(Employee)group[0].getPosition(); **//Class Cast Exception**

Η λύση - RTTI

- Για τη μέθοδο `toString` που υπάρχει και στις δύο κλάσεις, το πρόβλημα λύνεται αυτόματα.
- Χωρίς να κάνουμε `downcasting`.
`group[1].toString();`
`group[0].toString();`
- Αν βρει αντικείμενο της κλάσης `Human` καλεί την `toString` της `Human`. Αν βρει αντικείμενο της κλάσης `Employee` καλεί αυτόματα την αντίστοιχη `toString`.
- Run Time Type Identification – Καθορισμός τύπου την ώρα εκτέλεσης

Δομές αντικειμένων

- Σε ArrayList και Vector αποθηκεύουμε αντικείμενα διαφόρων κλάσεων που όλες κληρονομούν από την ίδια βασική κλάση.
- Η βασική κλάση έχει μεθόδους και οι παράγωγες κλάσεις τις υπερβαίνουν
- Όταν ανακτούμε ένα αντικείμενο από τη δομή το μετατρέπουμε στη βασική κλάση και καλούμε τις μεθόδους του.
- Ανάλογα με τον τύπο του αντικειμένου παίρνουμε και άλλη συμπεριφορά - **Πολυμορφισμός**

Πλεονεκτήματα του πολυμορφισμού

- Μπορούμε να ασχολούμαστε με τη γενική συμπεριφορά των αντικειμένων και να αφήνουμε τη συγκεκριμένη συμπεριφορά του καθενός να ορίζεται την ώρα εκτέλεσης
- Διευκολύνει την επέκταση. Καθώς τα μηνύματα κλήσης είναι ίδια (προς τη βασική κλάση) νέες κλάσεις μπορούν να δημιουργηθούν αρκεί να καθορίσουν το δικό τους τρόπο χειρισμού των μηνυμάτων.

Αφηρημένες κλάσεις – abstract classes

- Μια abstract κλάση βρίσκεται στην κορυφή μιας ιεραρχίας κλάσεων και συγκεντρώνει λειτουργίες.
- Οι υπόλοιπες κλάσεις της ιεραρχίας υλοποιούν τις λειτουργίες αυτές με το δικό τους τρόπο
- Δεν μπορούμε να φτιάξουμε αντικείμενα abstract κλάσεων μπορούμε όμως να έχουμε αναφορές σε abstract κλάσεις.
- Η δήλωση final σε μια κλάση της ιεραρχίας δεν επιτρέπει επιπλέον κληρονομικότητα
- Η δήλωση final σε μια μέθοδο μιας κλάσης δεν επιτρέπει επιπλέον πολυμορφισμό

Η κλάση Shape

- Η Shape παρέχει όλες τις κοινές λειτουργίες

```
public abstract class Shape {  
    protected int x, y, w, h;  
    protected int penWidth;  
    protected Color penColor, fillColor;  
    public int getWidth() {...}  
    public int getHeight() {...}  
    public void move(int newX, int newY) {...}  
    public void resize(int newW, int newH) {...}  
    public void setPenColor(Color newColor) {...}  
    public void setPenWidth(int newWidth) {...}  
    ... // κλπ.  
}
```

Η κλάση Shape

- Δεν υπάρχουν όμως μέθοδοι όπως draw ή getArea για να ζωγραφίζουν και να βρίσκουν το εμβαδό του σχήματος
- Όλες οι υπο-κλάσεις έχουν τις δικές τους εκδόσεις των μεθόδων αυτών
- Η Shape θα μπορούσε να παρέχει μια «ψεύτικη» υλοποίηση. **Αρκεί** όμως να τις αναφέρει ως abstract **χωρίς** να τις υλοποιεί
π.χ. `public abstract double getArea();`
- Οι υπο-κλάσεις **πρέπει** να παρέχουν τη δική τους πραγματική έκδοση
- Αν μια κλάση έχει abstract μεθόδους **πρέπει** να δηλωθεί abstract

Παράδειγμα (2)

```
public abstract class Employee {  
    ...  
    public abstract double getWeeklySalary(); // ορίζεται στις απόγονες  
}  
public final class Manager extends Employee {  
    private double weeklySalary;  
    public double getWeeklySalary( ) {return weeklySalary;}  
}  
public final class PieceWorker extends Employee {  
    private double wagePerPiece; // μισθός ανά τεμάχιο  
    private int quantity; //τεμάχια παραγωγής  
    public double getWeeklySalary( ) {return wagePerPiece*quantity;}  
}  
public final class HourlyWorker extends Employee {  
    private double wage; // μισθός ανά ώρα  
    private double hours; //ώρες εργασίας  
    public double getWeeklySalary( ) {return wage*hours;}  
}
```

Πλεονεκτήματα

- Δηλώνουν μια επιθυμητή λειτουργικότητα και αφήνουν στις κλάσεις να την ορίσουν

```
public interface Shape {  
    public abstract double area(); // calculate area  
    public abstract double volume(); // calculate volume  
    public abstract String getName();// return shape name  
}  
  
public class Triangle implements Shape {...}
```

- Υποχρεωτικά θα πρέπει να ορίσει τις μεθόδους της διεπαφής Shape
- Είναι ένας έμμεσος τρόπος να έχουμε πολλαπλή κληρονομικότητα λειτουργιών στη Java

Τελικοί (Final) μέθοδοι

- Οι “final” μεταβλητές γίνονται σταθερές
 - Ανατίθεται ακριβώς μία φορά και δε μπορεί να αλλάξει
- Οι “final” μέθοδοι δεν είναι overridable
 - Η κλάση γονέα τις ορίζει μία φορά και δεν ορίζονται ξανά στις υπο-κλάσεις
 - Όλες οι private μέθοδοι είναι έμμεσα τελικές (implicitly final)
 - Οι μέθοδοι δεν μπορούν να είναι μαζί abstract και final - **γιατί**;
 - Εξασφαλίζουμε ότι η συμπεριφορά διατηρείται και δεν μπορεί να τις αλλάξει κανείς στις υπο-κλάσεις

Final κλάσεις

- Οι “final” κλάσεις δεν κληρονομούνται
 - Όλοι οι μέθοδοι της γίνονται έμμεσα τελικές
 - Όταν θέλουμε να είμαστε σίγουροι ότι κανείς δεν θα τις κληρονομήσει

public final class String{}

- Οι final μέθοδοι και κλάσεις δεν χρησιμοποιούνται συχνά

Η δήλωση static

- Οι static μεταβλητές κληρονομούνται αλλά δεν αναπαράγονται στις υπο-κλάσεις
π.χ. Αν η κλάση Employee έχει μεταβλητή
static int numCreated
που μετράει το πλήθος των στιγμιότυπων που δημιουργήθηκαν, θα μετρά ΚΑΙ τα στιγμιότυπα των Manager κλπ.
- Οι static μέθοδοι δεν χρησιμοποιούν dynamic (late) binding. Ο compiler παράγει κάπως αποδοτικότερο κώδικα.
- Static κλάσεις υπάρχουν ως μέλη άλλων κλάσεων



Interfaces

Abstract class- Interface

- Για μια κλάση που δηλώνεται abstract δεν μπορούμε να φτιάξουμε αντικείμενα.
- Μπορούμε να δηλώσουμε κάποια λειτουργικότητα και κάποια βασικά χαρακτηριστικά που θα τα κληρονομήσουν οι απόγονες κλάσεις.
- Τις abstract κλάσεις που ορίζουν μόνο μεθόδους τις δηλώνουμε ως διεπαφές – interfaces
- Τα interfaces συγκεντρώνουν μόνο **δηλώσεις λειτουργικότητας**. Άλλες κλάσεις αναλαμβάνουν να υλοποιήσουν (***implement***) τις δηλώσεις αυτές

Interface

- Το interface είναι μια συλλογή από “υπογραφές” μεθόδων (δεν υπάρχουν στιγμιότυπα, ούτε υλοποιήσεις των μεθόδων)
- Περιγράφει πρωτόκολλο/συμπεριφορά αλλά όχι υλοποίηση
- Όλες οι μέθοδοι του είναι public και abstract (ποτέ static)
- Όλες οι μεταβλητές είναι static και final
- Μια κλάση υλοποιεί (implements) ένα interface

Παράδειγμα – Enumeration, Iterator

- `java.util.Enumeration`: είναι ένα interface. Περιγράφει μεθόδους για το ψάξιμο μέσα σε μια συλλογή

```
public interface Enumeration{
    boolean hasMoreElements();
    Object nextElement(); }
```
- Ένας iterator υλοποιεί αυτό το interface
- Οι κλάσεις `Vector`, `Hashtable`, `Set`, `Graph`, `Tree` κλπ. υλοποιούν το `Enumeration` ορίζοντας πώς θα ανταποκρίνεται η κάθε μέθοδος
- Μπορούμε να χρησιμοποιήσουμε το interface ως όνομα τύπου σε όσες κλάσεις υλοποιούν το interface.

```
void printAll(Enumeration e) {
    while(e.hasMoreElements())
        System.out.println(e.nextElement());}
```

Παράδειγμα: VectorEnumerator

```
class VectorEnumerator implements Enumeration{  
    private Vector vector;  
    private int count;  
    VectorEnumerator(Vector v) {  
        vector = v;  
        count = 0; }  
    public boolean hasMoreElements() {  
        return count < vector.size(); }  
    public Object nextElement() {  
        return vector.elementAt(count++);}  
}
```

Σύνταξη του interface

- ΟΛΕΣ οι μέθοδοι ενός interface πρέπει να υλοποιηθούν, αλλιώς η νέα κλάση θα πρέπει να οριστεί ως `abstract`
- Οι μέθοδοι του interface πρέπει να είναι `public`
- Μια κλάση μπορεί να υλοποιήσει πολλαπλά interfaces
`public class Shape implements Colorable, Printable { ...}`
- Πρέπει να προσέχουμε τα ονόματα των μεθόδων στα interfaces που θα συνδυαστούν να μη συμπίπτουν γιατί δημιουργούνται συγχύσεις.

Κληρονομικότητα Interfaces

- Τα interfaces μπορούν να επεκτείνουν αλλά interfaces

```
public interface Beepable {  
    public void beep(); }  
public interface VolumeControlled extends Beepable {  
    public int getVolume (int newVol);  
    public void setVolume( int newVol);  
    public void mute(); }
```

- Η κλάση είναι συμβατή με τον τύπο του interface. Κάνουμε upcasting σε τύπο interface όπως θα κάναμε σε μια abstract ή σε μια οποιαδήποτε κλάση

Interfaces ή abstract κλάσεις ?

- Παρόμοιες χρήσεις
 - Σχεδιάστηκαν για την ομαδοποίηση της συμπεριφοράς,
 - για να επιτρέπουν το upcasting,
 - για την εκμετάλλευση του πολυμορφισμού
- Μια κλάση μπορεί να υλοποιήσει πολλαπλά interfaces, αλλά έχει μόνο μια υπερ-κλάση
- Το interface δεν έχει καθόλου υλοποίηση
 - Είναι καλό, αν η ομοιότητα συμπεριφοράς είναι μόνο στο όνομα
 - Είναι κακό, αν υπάρχει κοινός κώδικας που θα μπορούσε να μεταφερθεί στην abstract κλάση (σε μια όχι abstract μέθοδο)
- Αν υπάρχει κοινός κώδικας → abstract κλάση, αλλιώς interface

Τα πεδία των interfaces

- Είναι static και final και μπορούν να χρησιμοποιηθούν για να ομαδοποιήσουν σταθερές

```
public interface Months {  
    int JANUARY = 1, ..., DECEMBER = 12; }
```

...

Από κάθε άλλη κλάση: Months.JANUARY

- Αρχικοποίηση γίνεται όταν αναφερθούμε για πρώτη φορά στο Interface.

```
public interface RandVals {  
    int rint = (int)(Math.random() * 10);}
```

...

RandVals.rint;

Εσωτερικά interfaces και κλάσεις

- Μέσα σε ένα interface (σε μία κλάση) μπορούμε να δηλώσουμε ένα άλλο interface (μια κλάση)
- Χρησιμοποιούνται
 - για να ομαδοποιήσουμε σχετικά μεταξύ τους interfaces ή κλάσεις (λειτουργικότητα), που δε χρησιμοποιούνται σε άλλο περιεχόμενο.
 - για να διαχωρίσουμε τη λειτουργικότητα σε μια κλάση από την ίδια την κλάση
- Παράδειγμα:
 - μια κλάση BinaryTree μπορεί να έχει τις μεθόδους για προσθήκη και αφαίρεση κόμβων.
 - Μέθοδοι που υλοποιούν κάποιο αλγόριθμο ταξινόμησης ή αναζήτησης κόμβων ομαδοποιούνται σε εσωτερική κλάση της BinaryTree. Διαχωρίζοντας έτσι τις λειτουργίες



Χρήσιμες μέθοδοι

Σύγκριση: Η μέθοδος equals

- Για να δουλέψουν οι προηγούμενοι μέθοδοι για λίστες με αντικείμενα δικών μας κλάσεων πρέπει στις κλάσεις μας να έχουμε μια μέθοδο equals π.χ.

```
public boolean equals(Object o){  
    Department d=(Department)o; // πιθανό να παράγει  
                                   //ClassCastException  
    if (this.id==d.getId() && this.name.equals(d.getName()) &&  
        this.numStudents==d.getNumStudents())  
        return true;  
    else  
        return false;  
}
```

Ταξινόμηση

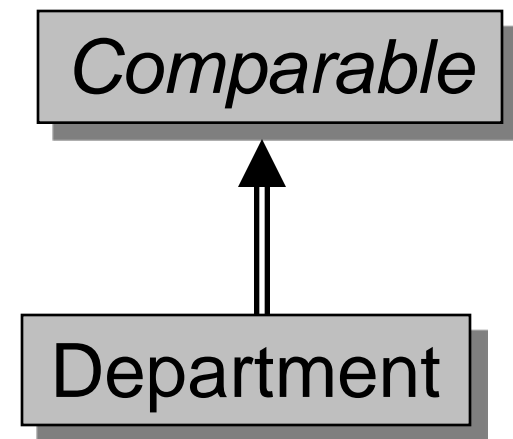
- Με ποιο τρόπο μπορώ να ταξινομήσω τα στοιχεία ενός πίνακα ή μιας λίστας; BubbleSort:


```
public void bubbleSort(int[] unsortedArray, int length) {  
    int temp, counter, index;  
    for(counter=0; counter<length-1; counter++) {  
        for(index=0; index<length-1-counter; index++) {  
            if(unsortedArray[index] > unsortedArray[index+1]) {  
                temp = unsortedArray[index];  
                unsortedArray[index] = unsortedArray[index+1];  
                unsortedArray[index+1] = temp;  
            }  
        }  
    }  
}
```

Διάταξη

- Η διάταξη στους ακεραίους είναι δεδομένη
- Τι γίνεται όμως με τις δικές μας κλάσεις;
- Πώς μπορούμε να ορίσουμε διάταξη στα αντικείμενά τους;

```
public interface Comparable  
{  
    int compareTo(Object o);  
}
```





```
public class Department implements Comparator{
    ...
    public int compareTo(Object o){
        Department d=(Department)o; // πιθανό να παράγει
                                     //ClassCastException
        if (this.numStudents>d.getNumStudents())
            return 1;
        else if (this.numStudents<d.getNumStudents())
            return -1;
        else
            return 0;
    }
}
```



Ταξινόμηση

```
Collections.sort(allDeps);
```

Ταξινομεί τα τμήματα με βάση τον αριθμό
σπουδαστών που έχουν

Χρησιμοποιεί την QuickSort