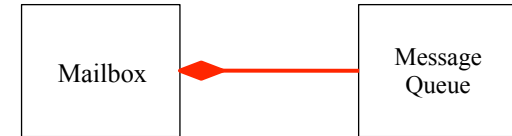


# Κληρονομικότητα

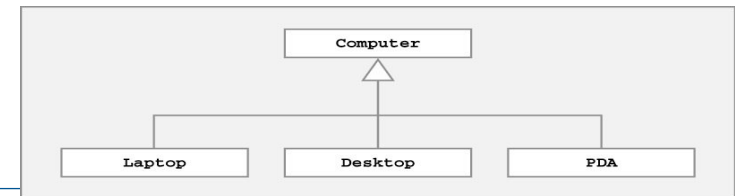
1

## Επαναχρησιμοποίηση Κλάσεων

- **Σύνθεση** (composition): σε μια κλάση εντάσσουμε ως στοιχεία χειριστήρια άλλων κλάσεων.



- **Κληρονομικότητα** (inheritance): επεκτείνουμε υπάρχουσες κλάσεις, είτε προσθέτοντάς τους καινούρια στοιχεία, είτε αλλάζοντας την λειτουργικότητα μεθόδων τους.



Μ. Δικαϊάκος, ΕΠΙΔ233

2

## Σύνθεση

```
class WaterSource {
    private String s;

    WaterSource() {
        System.out.println("WaterSource()");
        s = new String("Constructed");
    }

    public String toString() {
        return s;
    }
}
```

Η χρήση του **toString()**: όταν θέλουμε να μπορεί να γίνει μετατροπή ενός αντικειμένου μιας κλάσης σε συμβολοσειρά, δεν έχουμε παρά να ορίσουμε στην κλάση μια μέθοδο με όνομα **toString()**.

Μ. Δικαϊάκος, ΕΠΙΔ233

3

## Σύνθεση (παράδειγμα)

```
public class SprinklerSystem {
    private String valve1, valve2;
    WaterSource source;
    float f;

    void print() {
        System.out.println("valve1 =" + valve1);
        System.out.println("valve2 =" + valve2);
        System.out.println("f = " + f);
        System.out.println("source =" + source);
    }

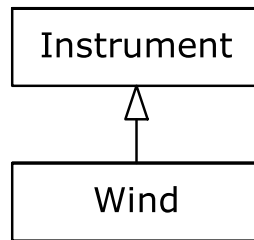
    public static void main (String[] args) {
        SprinklerSystem x = new SprinklerSystem();
        x.print();
    }
}
```

Μ. Δικαϊάκος, ΕΠΙΔ233

4

## Κληρονομικότητα (inheritance)

- Ορολογία:
  - Base class: βάση, κληροδότης, ή superclass, υπερκλάση.
  - Κλάση-κληρονόμος
- Απεικόνιση:



## Κληρονομικότητα (inheritance)

- Όλες οι κλάσεις της Java κληρονομούν από μια γενική κλάση-κληροδότη, την **Object**.
  - Υπάρχει επίσης η δυνατότητα να ορίζουμε κλάσεις ως κληρονόμους άλλων κλάσεων, πιο εξειδικευμένων από την **Object**.
  - Η κληρονομικότητα μιας κλάσης από κάποια κλάση-κληροδότη καθορίζεται με χρήση της λέξης-κλειδί **extends**.  
Π.χ.:
- **public class Detergent extends Cleanser**
  - Στην περίπτωση αυτή η νέα κλάση (Detergent) αυτομάτως «κληρονομεί» όλα τα δημόσια και τα φιλικά στοιχεία (πεδία δεδομένων και μέθοδοι) της βάσης της (Cleanser).

## Παράδειγμα Κληρονομικότητας

```
class Cleanser {
    private String s = "Cleanser";
    public void append(String a) { s += a;}
    public void dilute() { append("dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public void print() { System.out.println(s); }
    public static void main (String[] args) {
        Cleanser x = new Cleanser();
        x.dilute();
        x.apply();
        x.scrub();
        x.print();
    }
}
```

## Παράδειγμα Κληρονομικότητας (συνέχεια)

```
public class Detergent extends Cleanser{
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub();//Call base-class
    }
    // Add methods to the interface:
    public void foam() { append(" foam()"); }
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        x.print();
        Cleanser.main(args);
    }
}
```

## Παράδειγμα Κληρονομικότητας (συνέχεια)

- Η κλάση **Cleanser** έχει ορισμένες μεθόδους στην διαπροσωπεία της (interface): **append()**, **dilute()**, **apply()**, **scrub()**, **print()**.
- Η **Detergent**, επειδή προκύπτει από την **Cleanser** (μέσω κληρονομικότητας), υιοθετεί αυτόματα όλες αυτές τις μεθόδους στην διαπροσωπεία της : η κληρονομικότητα συνεπάγεται την **επαναχρησιμοποίηση** της διαπροσωπείας.
- Η κλάση-κληρονόμος μπορεί να **τροποποιήσει** την λειτουργικότητα μιας μεθόδου της κλάσης-κληροδότη, καλώντας ταυτόχρονα την αναλλοίωτη μέθοδο της κλάσης κληροδότη (με χρήση της λέξης-κλειδί **super**, η οποία παραπέμπει στην υπερκλάση-κληροδότη).
- Επιπλέον, η κλάση-κληρονόμος μπορεί να **επεκτείνει** την διαπροσωπεία της κλάσης-κληροδότη με πρόσθεση νέων μεθόδων.

## Αρχικοποιήσεις και Κληρονομικότητα

- Η κλάση-κληρονόμος έχει την ίδια διαπροσωπεία (interface) με την υπερκλάση της, επεκτεινόμενη ίσως με κάποιες καινούριες μεθόδους και πιθανώς με τροποποιημένες κάποιες από τις κληρονομημένες μεθόδους.
- Τι περιέχει το αντικείμενο που δημιουργείται σύμφωνα με μια κλάση-κληρονόμο;
  - Ένα **υποαντικείμενο** (subobject) της κλάσης-κληροδότη.
  - Το υποαντικείμενο αυτό είναι το ίδιο σαν να είχε δημιουργηθεί ανεξάρτητα από την κλάση-κληροδότη και **όχι εμμέσως** από την κλάση-κληρονόμο.
  - Ωστόσο το υποαντικείμενο αυτό είναι περιτυλιγμένο (wrapped) στο αντικείμενο της κλάσης-κληρονόμου.
  - Χρειάζεται προσοχή στην αρχικοποίηση του υποαντικειμένου.

## Αρχικοποιήσεις και Κληρονομικότητα

- Η κατάλληλη αρχικοποίηση του υποαντικειμένου, που αντιστοιχεί στην κλάση-κληροδότη, εξασφαλίζεται με την κατάλληλη κλήση του κατασκευαστή (constructor) της κλάσης-κληροδότη.
- Η κλήση προς τον κατασκευαστή γίνεται **υπόρρητα** από την Java, η οποία εισάγει **αυτομάτως** στο σώμα του κατασκευαστή της κλάσης-κληρονόμου μια κλήση προς τον κατασκευαστή εκείνο του κληροδότη, ο οποίος δεν επιδέχεται παραμέτρων.
- Η κλήση των κατασκευαστών σε μια σειρά κληρονομικότητας **ξεκινάει από τον πιο απόμακρο κληροδότη**, προχωρώντας προς τον τελευταίο κληρονόμο αφού:
  - Το αντικείμενο μιας κλάσης-κληροδότη πρέπει να έχει αρχικοποιηθεί πριν να μπορέσουν οι κατασκευαστές των κλάσεων-κληρονόμων να το χρησιμοποιήσουν.

## Παράδειγμα

```
class Art {
    Art() {
        System.out.println("→ Art");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("→ Draw");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("→ Cartoon");
    }
    public static void
        main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

```

class Art {
    Art() {
        System.out.println("→ Art");
    }
}

class Drawing extends Art {
    Drawing() {
        super();
        System.out.println("→ Draw");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        super();
        System.out.println("→ Cartoon");
    }

    public static void
        main(String[] args) {
        Cartoon x = new Cartoon();
    }
}

```

## Κληρονομικότητα και Κατασκευαστές

- Στο προηγούμενο παράδειγμα και για όλες τις κλάσεις στην σειρά κληρονομικότητας που εξετάσαμε, έχουμε προκαθορισμένους (default) κατασκευαστές χωρίς παραμέτρους.
- Επομένως, η προκαθορισμένη συμπεριφορά της JAVA να εισαγάγει υπόρρητα κλήσεις προς τους κατασκευαστές των κληροδοτών, χωρίς πέρασμα παραμέτρων, δεν δημιουργεί πρόβλημα.
- Σε διαφορετική περίπτωση (υπερφόρτωση κατασκευαστών, απουσία προκαθορισμένου κατασκευαστή), ο προγραμματιστής πρέπει να καθορίσει **ρητά** την κλήση προς τον κατασκευαστή της κλάσης-κληροδότη, περνώντας και τις κατάλληλες παραμέτρους.
- Η κλήση ενός κατασκευαστή της υπερκλάσης γίνεται με χρήση της μεθόδου **super()**.

```

class Game {
    Game(int i) { System.out.println("Game constr"); }
}

class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        System.out.println("BoardGame const");
    }
}

public class Chess extends BoardGame {
    Chess() {
        super(11);
        System.out.println("Chess constr");
    }
}

public static void main(String[] args) {
    Chess x = new Chess();
}
}

```

if not used, compiler looks for super()

## Σύνθεση και Κληρονομικότητα

```

public class PlaceSetting extends Custom {
    Spoon sp;
    Fork frk;
    Knife kn;
    DinnerPlate pl;
    PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
        frk = new Fork(i + 3);
        kn = new Knife(i + 4);
        pl = new DinnerPlate(i + 5);
    }
    public static void main (String[] args) {
        PlaceSetting x = new PlaceSetting(9);
    }
}

```

- Ο μεταφραστής μας εξαναγκάζει να αρχικοποιήσουμε τους κληροδοτές, αλλά εμείς πρέπει να φροντίσουμε για την αρχικοποίηση των αντικειμένων-στοιχείων.

## Quiz

```
class Instrument {
    Instrument() {}
    Instrument(int i) {}
}

public class Wind extends Instrument {
    Wind() {}
    Wind(int i) {
        // super();
        this();
    }
}

public static void main(String args[]) {
    Wind flute = new Wind(8);
}
```

Άλλη εντολή πριν την  
this() δεν επιτρέπεται

Μέσω της this(), καλείται  
ο κατασκευαστής Wind(), ο οποίος  
με τη σειρά του καλεί τον Instrument()

- Ποιός κατασκευαστής της **Instrument** θα κληθεί;
- Πόσες φορές;

## Απόκρυψη Ονομάτων

```
class Homer {
    char doh(char c) {
        System.out.println("doh(char)");
        return 'd';
    }
    float doh(float f) {
        System.out.println("doh(float)");
        return 1.0f;
    }
}
```

```
class Milhouse { }

class Bart extends Homer {
    void doh(Milhouse m) { }
}
```

```
public class Hide {
    public static void main(String[] args){
        Bart b = new Bart();
        b.doh(1); // doh(float) used
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
    }
}
```

Η υπερφόρτωση λειτουργεί ανεξαρτήτως κληρονομικότητας.

## Σύνθεση και Κληρονομικότητα

- **Σύνθεση :**
  - Η σύνθεση χρησιμοποιείται γενικότερα όταν χρειαζόμαστε τα χαρακτηριστικά μιας υπάρχουσας κλάσης και όχι την διαπροσωπεία της.
  - Αντιστοιχεί σε **has-a** σχέσεις ανάμεσα στις κλάσεις .
- **Κληρονομικότητα :**
  - Χρησιμοποιείται όταν θέλουμε να φτιάξουμε μια **ειδική περίπτωση** κάποιας κλάσης.
  - Αν μια κλάση B κληρονομεί από μια κλάση A , η σχέση τους συνοψίζεται ως εξής : Η νέα κλάση B είναι ένας τύπος της υπάρχουσας κλάσης A .
  - Αντιστοιχεί σε **is-a** σχέσεις ανάμεσα στις κλάσεις.

## Αντιπροσώπευση - delegation

```
public class SpaceShipControls {
    void up(int velocity) {}
    void down(int velocity) {}
    void left(int velocity) {}
    void right(int velocity) {}
    void forward(int velocity) {}
    void back(int velocity) {}
    void turboBoost() {}
}
```

```
public class SpaceShip extends SpaceShipControls {
    private String name;
    ...
}
```

```

public class SpaceShipDelegation{
    private String name;
    private SpaceShipControls controls =
        new SpaceShipControls();
    public SpaceShipDelegation(String name) {
        this.name = name;
    }

    public void back(int velocity) {
        controls.back(velocity);
    }
    ...
}

```

## Προσδιοριστής πρόσβασης: protected

```

import java.util.*;
class Villain {
    private int i;
    public Villain(int ii) { i = ii; }
    protected int read() { return i; }
    protected void set(int ii) { i = ii; }
    public int value(int m) { return m*i; }
}
public class Orc extends Villain {
    private int j;
    public Orc(int jj) { super(jj); j = jj; }
    public void change(int x) { set(x); }
}

```

- Η χρήση του **protected** σημαίνει ότι το αντίστοιχο στοιχείο μπορεί να χρησιμοποιηθεί μέσα στην βιβλιοθήκη στην οποία ανήκει, αλλά και στις κλάσεις που κληρονομούν από την κλάση του στοιχείου, όχι όμως σε άλλες άσχετες κλάσεις.

## Αναβάθμιση κ. κληρονομικότητα

```

import java.util.*;
class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}

class Wind extends Instrument {
    public static void main(String[] args){

        Wind flute = new Wind();
        Instrument.tune(flute);
    }
}

```

Upcasting

## Αναβάθμιση και κληρονομικότητα

- Η κληρονομικότητα συνεπάγεται ότι **οποιοδήποτε «μήνυμα» στέλνεται σε μια κλάση-κληροδότη, μπορεί να σταλεί και στις κλάσεις-κληρονόμους της.**
- Αυτό σημαίνει ότι **όλες** οι μέθοδοι που ανήκουν σε μια κλάση-κληροδότη **και για τις οποίες δίνονται δικαιώματα πρόσβασης σε άλλες κλάσεις**, ενυπάρχουν και στις κλάσεις-κληρονόμους της.
  - **Δεν** συμβαίνει απαραίτητα και το αντίστροφο
- Επομένως, οι μέθοδοι **play** και **tune** του παραδείγματός μας «υπάρχουν» και στα αντικείμενα τύπου **Wind**.

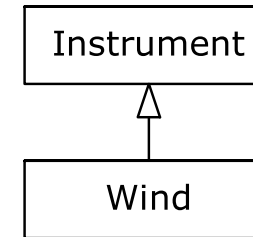
## Αναβάθμιση - upcasting

- Παρατηρείστε ότι η **tune** καλείται από την **Wind.main** με παράμετρο τύπου **Wind** χωρίς να παραπονεθεί ο μεταφραστής της Java.  

```
class Wind extends Instrument {  
    Wind flute = new Wind();  
    Instrument.tune(flute);  
}
```
- Τι είναι το **upcasting**;
  - Η αναβάθμιση ενός χειριστήριου τύπου **Wind** σε χειριστήριο τύπου **Instrument**.
- Η μετατροπή γίνεται αυτόματα .

## Αναβάθμιση

- Λόγω της κληρονομικότητας, ένα αντικείμενο τύπου **Wind** είναι **επίσης** αντικείμενο τύπου **Instrument**.
- Στη διαπροσωπεία (interface) του **Wind** υπάρχουν όλες οι μέθοδοι της διαπροσωπείας του **Instrument**.



## Quiz

```
package epl233;  
import java.util.*;  
class Instrument {  
    private static int k = 8;  
    void play() {  
        System.out.println("play instrument");  
    }  
    static void tune(Instrument i) {  
        System.out.println("Private k:" + k);  
        i.play();  
    }  
}  
class Wind extends Instrument {  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        flute.k = 9;  
        flute.tune(flute);  
    }  
}
```

Μήνυμα σφάλματος  
αφού δεν έχουμε  
πρόσβαση στην k

## Η χρήση του final

- Η λέξη **final** έχει διάφορες χρήσεις στην Java, αναλόγως των συμφραζομένων και αφορά σε:
  - Δεδομένα.
  - Μεθόδους
  - Κλάσεις
- Γενικά σηματοδοτεί κάτι το οποίο δεν μπορεί ν' αλλάξει.
- Δεδομένα final:**
  - Με την **final** μπορούμε να ορίζουμε **σταθερές** στην **μεταφράση** (compile-time constants), οι οποίες ανήκουν σε **αρχέγονους τύπους**.
  - Μπορούμε επίσης να ορίζουμε σταθερές στην **εκτέλεση** (run-time constants), οι οποίες αρχικοποιούνται κάποια στιγμή στον χρόνο εκτέλεσης και δεν αλλάζουν έκτοτε.
  - Στην περίπτωση χειριστηρίων που καθορίζονται ως **final**, τα ίδια τα χειριστήρια γίνονται «σταθερές», όχι όμως και τα περιεχόμενα των αντικειμένων στα οποία παραπέμπουν.
  - Πεδία που ορίζονται σαν **final** και **static** αντιστοιχούν σε μια μοναδική θέση μνήμης, τα περιεχόμενα της οποίας δεν μπορούν να αλλάξουν.

## Παράδειγμα

```
class Value { int i = 1; }
public class FinalData {
    final int i1 = 9;
    static final int VAL_TWO = 99;
    public static final int VAL_THREE=39;
    final int i4 =(int)(Math.random()*20);
    static final int i5 = (int)(Math.random()*20);
    Value v1 = new Value();
    final Value v2 = new Value();
    static final Value v3 = new Value();
    // Arrays:
    final int[] a = { 1, 2, 3, 4, 5, 6 };
```

Compile-time constants

Typical public constant

Run-time constants

## Τελικές Παράμετροι

```
class Gizmo { public void spin() {} }
public class FinalArguments {
    void with(final Gizmo g) {
        g = new Gizmo();
        g.spin();
    }
    void without(Gizmo g) {
        g = new Gizmo();
        g.spin();
    }
    void f(final int i) { i++; }
    int g(final int i) { return i + 1; }
    public static void main(String[] args) {
        FinalArguments bf = new FinalArguments();
        bf.without(null);
        bf.with(null); } }
```

Προκύπτει σφάλμα: παράμετροι οι οποίες είναι final δεν μπορούν να αλλαχθούν.

## Κενές Τελικές (Blank finals)

- Η Java επιτρέπει τον ορισμό πεδίων σαν **final**, χωρίς να γίνεται αρχικοποίηση τους στο σημείο του ορισμού.
- Στην περίπτωση αυτή τα πεδία-σταθερές **πρέπει** να αρχικοποιηθούν μέσα στον κατασκευαστή της αντίστοιχης κλάσης – διαφορετικά ο μεταφραστής διαμαρτύρεται.
- Έτσι, μας δίνεται η δυνατότητα να έχουμε διαφορετικές σταθερές μέσα σε διαφορετικά αντικείμενα της ίδιας κλάσης .

## Τελικές Μέθοδοι

- Υπάρχουν δύο λόγοι για τους οποίους δηλώνουμε μια μέθοδο σαν **final**:
  - Για να αποτρέψουμε την αναίρεση (overwriting) της μεθόδου μέσα από κάποια κλάση που την κληρονομεί , και να διατηρήσουμε έτσι το νόημά της.
  - Για λόγους απόδοσης: ο κώδικας μεθόδων **final** μπορεί να «ενσωματώνεται» (inline) από τον μεταφραστή στο κάθε σημείο κλήσης της μεθόδου.
  - Κάθε **ιδιωτική μέθοδος** σε μια κλάση είναι **υπορρήτως τελική** – αφού δεν επιτρέπεται πρόσβαση προς αυτήν από κάποιον κληρονόμο.

## Ιδιωτικές και Τελικές Μέθοδοι

```
public class Test {  
    final void foo(int i) { }  
}  
  
public class Test1 extends Test {  
    void foo(int j) { }  
    void foo(char c) { }  
}
```

- Προκύπτει σφάλμα στην μετάφραση (που και γιατί);

```
public class Test {  
    private void foo(int i) { }  
}  
  
public class Test1 extends  
    Test {  
    void foo(int j) { }  
    void foo(char c) { }  
}
```

- Δεν προκύπτει σφάλμα στην μετάφραση.

## Τελικές Κλάσεις

- Η δήλωση μιας κλάσης **Foo** ως **final**, υποδηλώνει ότι η κλάση αυτή **δεν** μπορεί να κληροδοτηθεί σε άλλες κλάσεις.
- Κατ' αυτόν τον τρόπο, δηλαδή, αποτρέπουμε την κληρονομικότητα από την **Foo**.

## Αρχικοποίηση και Κληρονομικότητα

```
class Insect {  
    int i = 9; int j;  
    Insect() { prt("i = " + i + ", j = " + j); j=39; }  
    static int x1 = prt("static Insect.x1 initialized");  
    static int prt(String s) { System.out.println(s); return 47; }  
}  
  
public class Beetle extends Insect {  
    int k = prt("Beetle.k initialized");  
    Beetle() {  
        prt("k = " + k);  
        prt("j = " + j);  
    }  
    static int x2 = prt("static Beetle.x2 initialized");  
    public static void main(String[] args){  
        prt("Beetle constructor");  
        Beetle b = new Beetle(); }}  
}
```

## Έξοδος Παραδείγματος

```
static Insect.x initialized  
Static Beetle.x2 initialized  
Beetle constructor  
i = 9, j = 0  
Beetle.k initialized  
k = 47  
i = 39
```

- Βήματα στην εκτέλεση του **java Beetle**:
  - Φόρτωση της κλάσης **Beetle**
  - Αναζήτηση και Φόρτωση της ιεραρχίας των κληροδοτών της **Beetle** (δηλαδή της **Insect**).
  - Στατική αρχικοποίηση της **Insect** και μετά της **Beetle**.
  - Δημιουργία αντικειμένων και αρχικοποίηση πεδίων τους (σε **null**).
  - Κλήση κατασκευαστών
  - Αρχικοποίηση instance variables
  - Εκτέλεση εντολών κατασκευαστή

## Πολλαπλή δήλωση της `main`

- Η μέθοδος **`main`** μπορεί να οριστεί σε καθεμιά από τις κλάσεις μας.
- Αν σε ένα αρχείο **`Foo.java`** υπάρχουν πολλές κλάσεις και καθεμιά τους έχει ορισμένη μια διαφορετική **`main`**, η κλήση **`java Foo`** θα προκαλέσει την εκτέλεση της **`main`** που ορίζεται στην δημόσια κλάση του αρχείου (την `Foo`).
- Αν μέσα στο **`Foo.java`** ορίζεται η κλάση **`Goo`**, η οποία έχει μια δημόσια μέθοδο **`main`**, τότε η κλήση **`java Goo`** θα εκτελέσει την μέθοδο **`Goo.main()`**, έστω κι αν η **`Goo`** δεν είναι δημόσια.
- Ο ορισμός μεθόδων **`main`** σε κάθε κλάση διευκολύνει ενίοτε τον έλεγχο της ορθότητας των κλάσεων.