Αντικειμενοστραφής Προγραμματισμός

Β' εξάμηνο – Τμήμα Πληροφορικής – Ιόνιο Πανεπιστήμιο

Ενότητα 7

Αφηρημένες Κλάσεις · Διεπαφές · Απαριθμήσεις

Δημήτρης Ρίγγας

Μηχανικός Η/Υ & Πληροφορικής, MSc, PhD

Ε.Δι.Π. Τμήματος Πληροφορικής – Ιόνιο Παν/μιο

riggas@ionio.gr

Αφηρημένες Κλάσεις

Abstract Classes

Αφηρημένες Κλάσεις (Abstract Classes)

Ας θυμηθούμε την Κληρονομικότητα

- Υπο-κλάση κληρονομεί και επεκτείνει υπερ-κλάση
- Σε κάποιες περιπτώσεις **δεν επιθυμούμε** να μπορεί να δημιουργηθεί αντικείμενο της υπερ-κλάσης
- Θέλουμε τη βασική κλάση **μόνο** για χρήση από τις υπο-κλάσεις

Δύο ρόλοι κλάσεων:

Κλάση	Χαρακτηριστικό	Αντικείμενα
Βασική (υπερ-κλάση)	**abstract**	❌ Δεν δημιουργούνται
Υπο-κλάση	**concrete**	✅ Δημιουργούνται

Πώς Ορίζεται η Abstract Κλάση

Χρήση της λέξης-κλειδί `abstract`:

```
// Αφηρημένη υπερ-κλάση
public abstract class MyClass {
    // ...
}

// Συμπαγής υπο-κλάση
public class MySpecificSubClass extends MyClass {
    // ...
}
```

✓ Μια κλάση μπορεί να είναι `abstract` ακόμα και αν όλες οι μέθοδοί της έχουν υλοποίηση — αρκεί να μη θέλουμε να δημιουργούνται αντικείμενά της.

Παράδειγμα: Abstract Class `Post`

```
public abstract class Post {
    private String username;
    protected long timestamp;
    private int likes;

    public Post(String username) {
        this.username = username;
        this.timestamp = Long.parseLong(
            (new SimpleDateFormat("yyyyMMddHHmm")).format(new Date()));
    }
    /* οι μέθοδοι παραλείπονται */
}

public class TextPost extends Post {
    /* παραλείπονται */
}
```

✅ `Post p = new TextPost("George", "Hello..");` — επιτρέπεται

❌ `Post p = new Post("George");` — σφάλμα μεταγλωττιστή

Αφηρημένες Μέθοδοι

Μια κλάση μπορεί να **δηλώνει** μεθόδους **χωρίς να τις υλοποιεί**:

- Δηλώνει μόνο την **υπογραφή** (signature) — χωρίς σώμα `{}`
- Παρέχει ένα γενικό interface χωρίς να καθορίζει πώς υλοποιείται
- Αυτή η κλάση **είναι υποχρεωτικά** `abstract`

Κανόνες:

	Abstract κλάση	Concrete υπο-κλάση
Υλοποίηση	Ελλιπής	Πλήρης
Μη-υλοποιημένες μέθοδοι	Επιτρέπονται	Απαγορεύονται*

√ Αν μια υπο-κλάση δεν υλοποιεί **όλες** τις abstract μεθόδους, πρέπει και αυτή να δηλωθεί `abstract`.

Πώς Ορίζονται οι Abstract Μέθοδοι

```java
// Αφηρημένη υπερ-κλάση με ελλιπή υλοποίηση
public abstract class MyClass {
    public abstract void myMethod();  // ← προσοχή: ";" αντί για "{}"
}

// Συμπαγής υπο-κλάση – πρέπει να υλοποιήσει myMethod()
public class MySpecificSubClass extends MyClass {
    @Override
    public void myMethod() {
        // do something
    }
}
```

Σημαντικό:

- Η abstract κλάση μπορεί να περιέχει και **concrete** (υλοποιημένες) μεθόδους
- Η υπο-κλάση χρησιμοποιεί `@Override` για σαφήνεια (καλή πρακτική)

Παράδειγμα: Abstract Μέθοδος `render()`

Θέλουμε διαφορετικές αποδόσεις `Post` ανά μέσο (HTML, Android, κ.λπ.):

```java
public abstract class Post {
    /* παραλείπονται */
    public abstract String render();
}

public abstract class PhotoPost extends Post {
    /* παραλείπονται */
    // Η render() είναι ήδη κληρονομημένη και abstract εδώ, από την Post
}

public class HTMLPhotoPost extends PhotoPost {
    @Override
    public String render() {
        return "<div>\n"
            + "<img src='" + super.getPhoto() + "'/>\n"
            + "<span>" + super.toString() + "</span>\n"
            + "</div>";
    }
}
```

Περιορισμός Κληρονομικότητας: `final`

Χρήση `final` για να αποτραπεί override μεθόδου σε υπο-κλάση:

```
public abstract class Post{
    /* παραλείπεται */
    public final void increaseLikes() {
        this.likes++;
    }
}

public class PhotoPost extends Post {
    // ❌ Σφάλμα μεταγλωττιστή!
    // increaseLikes() is final in Post
    public final void increaseLikes() {
        super.increaseLikes();
        super.increaseLikes();  // "cheating not allowed!"
    }
}
```

✓ Το `final` μπορεί να εφαρμοστεί και σε **κλάσεις** (`final class`) για να αποτραπεί εξολοκλήρου η κληρονομικότητα (πχ `String`, `Integer`).

`final` vs `sealed` Classes

	`final class`	`sealed class`
Κληρονομικότητα	❌ Καμία	✅ Μόνο από `permits` λίστα
Instantiation	✅ Επιτρέπεται	Εξαρτάται αν είναι και abstract
Έλεγχος switch	❌	✅ (Java 21+)
Ευελιξία	Καμία	Ελεγχόμενη

```java
// final – κανείς δεν μπορεί να κληρονομήσει
public final class ImmutablePoint {
    private final int x, y;
    // ...
}
// ImmutablePoint δεν μπορεί να επεκταθεί από κανέναν

// sealed – μόνο συγκεκριμένες κλάσεις επιτρέπονται
public abstract sealed class Shape
    permits Circle, Rectangle, Triangle { }

public final class Circle    extends Shape { }  // δεν επεκτείνεται περαιτέρω
public non-sealed class Rectangle extends Shape { }  // ελεύθερη κληρονομικότητα
```

✓ `final` = **«κανένας»** — `sealed` = **«μόνο αυτοί»**

Διεπαφές

Interfaces

Διεπαφές (Interfaces)

Καλύπτουν την ανάγκη διαφορετικών τμημάτων ενός προγράμματος να **συμφωνήσουν σε ένα «συμβόλαιο»** αλληλεπίδρασης.

- Κάθε τμήμα που υλοποιεί ένα interface δεν χρειάζεται να γνωρίζει πώς λειτουργούν τα υπόλοιπα
- Είναι **αναφερόμενος τύπος** (reference type), όπως η κλάση
- **Δεν** μπορούν να δημιουργηθούν αντικείμενα (`instances`) διεπαφών
- Μπορούν να **υλοποιηθούν** (`implements`) από κλάσεις ή να **επεκταθούν** (`extends`) από άλλες διεπαφές

Χαρακτηριστικά (κλασικά — έως Java 7):

- Όλες οι μέθοδοι είναι εξ ορισμού `abstract` και `public`
- Δεν περιλαμβάνουν μεταβλητές, μόνο **σταθερές** (`public static final`)
- Υποστηρίζουν **πολλαπλή κληρονομικότητα** (μια κλάση μπορεί να υλοποιεί πολλά interfaces)

Πώς Ορίζεται ένα Interface

```java
// Ορισμός interface
public interface MyInterface {
    public void myMethod();  // ← δεν υπάρχουν {} αλλά ;
}

// Υλοποίηση interface από κλάση
public class MyClass implements MyInterface {
    @Override
    public void myMethod() {
        /* do something */   // εδώ παρέχεται η υλοποίηση
    }
}
```

✓ Αν μια κλάση δεν υλοποιεί **όλες** τις μεθόδους που ορίζει το interface, τότε πρέπει να δηλωθεί `abstract`.

Πολλαπλά interfaces:

```java
public class MyClass implements InterfaceA, InterfaceB, InterfaceC {
    // ...
}
```

Παράδειγμα: Interface `Likable`

```java
public interface Likable {
    public void increaseLikes();
    public int getLikes();
    /** @returns -1 αν other πιο likable, 1 αν this, 0 αλλιώς */
    public int isMoreLikableThan(Likable other);
}

public abstract class Post implements Likable {
    private int likes;
    /* παραλείπονται */
    public void increaseLikes()  { this.likes++; }
    public int getLikes()        { return this.likes; }

    public int isMoreLikableThan(Likable other) {
        if (this.getLikes() == other.getLikes()) return 0;
        else if (this.getLikes() < other.getLikes()) return -1;
        return 1;
    }
}
```

Χρήση Interface ως Τύπο

Όταν ορίζουμε ένα interface, ορίζουμε ένα νέο **αναφερόμενο τύπο**:

```
// ✅ Επιτρέπεται – PhotoPost υλοποιεί το Likable
Likable lk = new PhotoPost(…);

// ❌ Δεν επιτρέπεται – δεν δημιουργούνται αντικείμενα interface
Likable lk = new Likable();
```

```
public void showMostLikable() {
    Likable mostLikable = (Likable) posts.getFirst();
    if (mostLikable == null) return;
    for (Post p : posts) {
        Likable lk = (Likable) p;
        if (mostLikable.isMoreLikableThan(lk) < 0)
            mostLikable = lk;
    }
    System.out.println("Most likable:\n" + mostLikable);
}
```

Διεπαφές ως Γενικοί Τύποι

Παράδειγμα: `java.lang.Comparable<T>` 🔗

```
public interface Comparable<T> {
    public int compareTo(T o);
}

public class Order implements Comparable<Order> {
    // …
    @Override
    public int compareTo(Order o) {
        if (o == null) throw new NullPointerException("Cannot compare to null");
        return (int)(this.netValue * this.quantity * (1.0 - this.discount)
                - o.netValue  * o.quantity  * (1.0 - o.discount));
    }
}
```

√ Αντικείμενα που υλοποιούν `Comparable` μπορούν να εισαχθούν σε δομές όπως `TreeSet` και να διατηρούνται **αυτόματα ταξινομημένα**

Static Μέθοδοι σε Interfaces (Java ≥ 8)

Από την έκδοση 8 της Java επιτρέπονται **υλοποιημένες static μέθοδοι** σε interfaces:

```java
public interface Likable {
    public void increaseLikes();
    public int getLikes();
    public int isMoreLikableThan(Likable other);

    // ✅ Νέο από Java 8
    public static double average(Likable[] objects) {
        double sum = 0;
        for (Likable l : objects) sum += l.getLikes();
        return objects.length > 0 ? sum / objects.length : 0;
    }
}

// Κλήση επί του ονόματος της διεπαφής:
double avg = Likable.average(posts);
```

```java
// ✅ Σωστό – κλήση μέσω του ονόματος της διεπαφής
double avg = Likable.average(posts);

// ❌ Σφάλμα μεταγλωττιστής – δεν κληρονομείται
double avg = PhotoPost.average(posts);  // PhotoPost implements Likable

// ❌ Σφάλμα – ούτε μέσω instance
PhotoPost p = new PhotoPost(…);
p.average(posts);
```

√ Static μέθοδοι δεν κληρονομούνται από τις κλάσεις που υλοποιούν το interface.

Default Μέθοδοι σε Interfaces (Java ≥ 8)

Ήρθαν για να επεκτείνουν interfaces **χωρίς να σπάνε παλιότερο κώδικα**:

```
public interface Likable {
    public void increaseLikes();
    public int getLikes();
    public int isMoreLikableThan(Likable other);

    // ✅ Νέο από Java 8 – default υλοποίηση
    public default boolean isMostLikable(Likable[] objects) {
        int topObjectLikes = 0;
        for (Likable l : objects)
            if (topObjectLikes < l.getLikes())
                topObjectLikes = l.getLikes();
        return getLikes() > topObjectLikes;
    }
}
```

- Κλάσεις που υλοποιούν το interface **υιοθετούν** αυτόματα τη default υλοποίηση
- Μπορούν να την κάνουν **override** αν χρειαστεί

Private Μέθοδοι σε Interfaces (Java 9+)

Επιτρέπονται **private μέθοδοι** σε interfaces για επαναχρησιμοποίηση κώδικα μεταξύ `default` και `static` μεθόδων, χωρίς έκθεση στις υλοποιούσες κλάσεις:

```
public interface Likable {
    public void increaseLikes();
    public int getLikes();

    public default boolean isMostLikable(Likable[] objects) {
        return getLikes() >= findMax(objects);  // ← χρήση του κοινού helper
    }

    public static double average(Likable[] objects) {
        double sum = 0;
        for (Likable l : objects) sum += l.getLikes();
        return objects.length > 0 ? sum / objects.length : 0;
    }

    public static Likable mostLikable(Likable[] objects) {
        int max = findMax(objects);             // ← και εδώ χρήση του κοινού helper
        for (Likable l : objects)
            if (l.getLikes() == max) return l;
        return null;
    }

    private static int findMax(Likable[] objects) {   // ← private static, όχι instance
        int max = 0;
        for (Likable l : objects)
            if (max < l.getLikes()) max = l.getLikes();
        return max;
    }
}
```

Απαριθμήσεις

Enumerations

Νέοι Τύποι με Κλειστό Σύνολο Τιμών

Οι τύποι δεδομένων έχουν συγκεκριμένο σύνολο τιμών (πχ `byte ≤ 127`).

Τι γίνεται αν θέλουμε δικούς μας τύπους με πεπερασμένες τιμές;

Παραδείγματα τιμών
Ημέρες εβδομάδας: MONDAY, TUESDAY, …, SUNDAY
Μήνες: JANUARY, FEBRUARY, …
Κατευθύνσεις: NORTH, SOUTH, EAST, WEST
Βαρύτητα πλανητών: MERCURY, VENUS, EARTH, …

Ο τύπος που ορίζεται από τον χρήστη με **σταθερό σύνολο τιμών** ονομάζεται:

“ **Απαριθμούμενος τύπος δεδομένων** (enumerated data type / `enum`) ”

Γιατί να Χρησιμοποιούμε Enums;

- **Ασφάλεια τύπων** — μια μεταβλητή τύπου `Day` δέχεται μόνο τιμές `MONDAY` … `SUNDAY`, τίποτα άλλο
- **Αυτο-τεκμηρίωση** — `WEDNESDAY` αντί για το «μαγικό νούμερο» `2`
- **Αποτρέπουν κατάχρηση** — αν χρησιμοποιούσατε `int` ή `String`, ένας χρήστης θα μπορούσε να χρησιμοποιήσει τον τελεστή `+`
- **Ενσωμάτωση σε collections** — `TreeSet<Day>`, `EnumMap<Day, …>`
- **Περιορισμός παραμέτρων μεθόδων** — μόνο οι τιμές της απαρίθμησης γίνονται δεκτές

```
// ✅ Ο μεταγλωττιστής ελέγχει τον τύπο
public void setWorkDay(Day d) { … }
setWorkDay(Day.MONDAY);     // OK
setWorkDay("Monday");       // ❌ Σφάλμα μεταγλωττιστή
setWorkDay(1);              // ❌ Σφάλμα μεταγλωττιστή
```

Πώς Ορίζεται ένα Enum

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
}
```

Κανόνες ονοματοδοσίας:

- Το όνομα του τύπου με **κεφαλαίο** αρχικό: `Day`, `MediaType`
- Οι σταθερές με **κεφαλαία**: `MONDAY`, `DVD`
- Οι σταθερές **δεν είναι** `String` — δεν εσωκλείονται σε εισαγωγικά

Σχέση με κλάσεις:

✓ Οι απαριθμήσεις είναι ένας **ειδικός τύπος κλάσης** στη Java — οι `public` απαριθμήσεις ορίζονται σε αρχείο ίδιο με το όνομά τους (`Day.java`)

Ορισμός & Πρόσβαση σε Τιμές

```
public enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

public class DaysOfTheWeek {
    public static void main(String[] args) {

        // values() – επιστρέφει πίνακα με όλες τις τιμές
        for (Day d : Day.values()) {
            System.out.println(d);
        }

        // Δήλωση μεταβλητής τύπου enum
        Day firstDayOfTheWeek = Day.MONDAY;
        System.out.println("The first day of the week is " + firstDayOfTheWeek);
        // Εκτυπώνει: The first day of the week is MONDAY
    }
}
```

- Η `toString()` επιστρέφει αυτόματα το όνομα της σταθεράς (πχ `"MONDAY"`)
- Η `values()` επιστρέφει τις τιμές **με τη σειρά που δηλώθηκαν**

Πεδία & Μέθοδοι σε Enum

```
public enum MediaTypes {
    DVD("DVD Optical Disk"),
    OGG("OGG Streaming Audio"),
    BOOK("Printed Text Media");           // ← `;` εδώ

    private final String description;

    MediaTypes(String descr) {            // ← constructor: private ή package-private
        this.description = descr;
    }

    public String getDescription() {
        return this.description;
    }

    public static void main(String[] args) {
        for (MediaTypes mt : MediaTypes.values()) {
            System.out.println(mt + "\t" + mt.getDescription());
        }
        // new MediaTypes("MP4"); ← ❌ enum types may not be instantiated
    }
}
```

Από String σε Τιμή Enum: `valueOf()`

```java
import java.util.Scanner;

public enum Planets {
    MERCURY, VENUS, EARTH, MARS, JUPITER,
    SATURN, URANUS, NEPTUNE, PLUTO;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Type a planet name: ");
        String s1 = sc.nextLine();

        // valueOf() – μετατρέπει String σε τιμή enum
        Planets p1 = Planets.valueOf(s1.toUpperCase()); // ← προσοχή στα κεφαλαία!

        System.out.println("Planet " + p1 + " is the "
            + (p1.ordinal() + 1) + " planet in our solar system");
    }
}
```

✓ Αν η τιμή δεν υπάρχει στην απαρίθμηση, η `valueOf()` ρίχνει `IllegalArgumentException`.

Σύγκριση Τιμών Enum

Με `ordinal()` — επιστρέφει τη θέση (0-based) της σταθεράς:

```
Planets p1 = Planets.valueOf(s1.toUpperCase());
Planets p2 = Planets.valueOf(s2.toUpperCase());

if (p2.ordinal() < p1.ordinal()) {
    Planets temp = p2;
    p2 = p1;
    p1 = temp;
}
System.out.println(p1 + " is closer to Sun than " + p2);
```

Άλλες μέθοδοι σύγκρισης:

Μέθοδος	Επιστρέφει	Χρήση
`ordinal()`	`int` (θέση, 0-based)	Σύγκριση θέσης
`compareTo(other)`	`int` (διαφορά ordinal)	Ταξινόμηση
`==`	`boolean`	Ισότητα τιμών
`name()`	`String`	Το όνομα ως String

Switch Expressions με Enums (Java ≥ 14)

```java
public enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

// Κλασσικό if – χρειάζεται qualifier, πχ Day.MONDAY
Day day = Day.MONDAY;
if (day == Day.MONDAY) { … }

// Παλιό switch statement – δεν χρειάζεται qualifier:
int numLetters;
switch (day) {
    case MONDAY: case FRIDAY: case SUNDAY: numLetters = 6; break;
    case TUESDAY:                          numLetters = 7; break;
    default:                               numLetters = 8;
}

// ✅ Νέο switch expression (Java 14+) – δεν χρειάζεται qualifier:
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};  // ← ο μεταγλωττιστής ελέγχει αν καλύπτονται ΟΛΕΣ οι τιμές!
    // Αν δεν καλυφθούν όλες οι τιμές ενός enum, ο **μεταγλωττιστής** εκδίδει σφάλμα.
    // ή εναλλακτικά προσθέτουμε με τελευταία default case
```

Περισσότερα Παραδείγματα

Συνδυασμός κλάσεων, κληρονομικότητας, διεπαφών, απαριθμήσεων και πολυμορφισμού

Abstract Super-class: `Person`

```
public abstract class Person {
    private String name;

    public Person(String name)     { this.name = name; }
    public void goodmorning()      { System.out.println(name + " says Goooodmorning!..."); }
    public void goodnight()        { System.out.println(name + " says Goodnight.."); }
    public String getName()        { return name; }

    public abstract void logDay();  // ← κάθε υπο-κλάση ορίζει τη δική της μέρα
}
```

- ✕ `Person p = new Person("Dimitris");` — δεν επιτρέπεται
- Κάθε συμπαγής υπο-κλάση **πρέπει** να υλοποιήσει την `logDay()`

Interface: `Employee`

```java
public interface Employee {
    public void goToWork();
    public void doTheWork();
    public void departFromWork();
}
```

- ❌ `Employee e = new Employee();` — δεν επιτρέπεται
- Απλά περιγράφει τι μπορεί να κάνει οποιοσδήποτε `Employee`
- Κάθε κλάση που κάνει `implement` αυτό το interface πρέπει (για να είναι concrete) να παρέχει υλοποιήσεις

Enum: `LecturerWork`

```
public enum LecturerWork {
    TEACH ("Be inspiring"),
    RESEARCH ("Appear thoughtful");

    private final String descr;
    LecturerWork(String d) { this.descr = d; }
    public String getDescription() { return this.descr; }
}
```

- ❌ `LecturerWork lw = new LecturerWork();` — δεν επιτρέπεται
- Απλά περιγράφει ένα κλειστό σύνολο τιμών

Concrete Sub-class: `Lecturer`

```
public class Lecturer extends Person implements Employee {
    private LecturerWork work;
    public Lecturer(String name) { super(name); }

    public void goToWork()      { System.out.println(getName() + " goes to university."); }
    public void doTheWork()     {
        int max = work.values().length;
        this.work = (work.values())[(new Random()).nextInt(max)];
        System.out.println(getName() + " current activity: " + work.getDescription());
    }
    public void departFromWork(){ System.out.println(getName() + " departs from university."); }

    @Override
    public void logDay() {
        this.goodmorning();
        this.goToWork();
        for (int i = 0; i < 5; i++) this.doTheWork();
        this.departFromWork();
        this.goodnight();
    }
}
```

- ✅ `Person p = new Lecturer("Dimitris");`
- ✅ `Employee e = new Lecturer("Dimitris");`
- ❌ `Lecturer l = new Person("Dimitris");`

Interface: `Parent`

```
public interface Parent {
    public void dropOffKids(String where);
    public void pickUpKids();
    public void attendParentsMeeting();
}
```

Sub-class: `LecturerWithKids`

```java
public class LecturerWithKids extends Lecturer implements Parent {
    private String whereAreTheKids = "home";
    public LecturerWithKids(String name) { super(name); }

    public void dropOffKids(String where) {
        System.out.println(getName() + " drops off kids at " + where);
        this.whereAreTheKids = where;
    }
    public void pickUpKids() {
        System.out.println(getName() + " picks up kids from " + whereAreTheKids);
        this.whereAreTheKids = "home";
    }
    public void attendParentsMeeting() {
        System.out.println(getName() + " attends parents meeting");
    }
    @Override
    public void logDay() {
        this.goodmorning();
        this.dropOffKids("school");
        this.goToWork();
        for (int i = 0; i < 5; i++) this.doTheWork();
        this.departFromWork();
        this.pickUpKids();
        this.goodnight();
    }
}
```

- ✅ `Lecturer l = new LecturerWithKids("Dimitris");`
- ✅ `Parent p = new LecturerWithKids("Dimitris");`

Αναφορά σε Αντικείμενα – Πολυμορφισμός

```
import java.util.*;
public class Main {
    private static Set<Person> people  = new HashSet<>();
    private static Set<Parent> parents = new HashSet<>();

    public static void main(String[] args) {
        Person p1 = new LecturerWithKids("Dimitris");
        people.add(p1);
        parents.add((Parent) p1);          // cast – LecturerWithKids implements Parent

        Person p2 = new ProgrammerWithKids("George");
        people.add(p2);
        parents.add((Parent) p2);

        Person p3 = new Lecturer("Kostas");
        people.add(p3);
        // parents.add((Parent) p3);       // ❌ Lecturer δεν implements Parent

        for (Person p : people)  p.logDay();
        for (Parent p : parents) ((Person) p).logDay();  // ← ευθύνη προγραμματιστή!
    }
}
```

Όροι που είδαμε

Έννοια	Χρήση	Instantiation
`abstract class`	Βασική κλάση χωρίς πλήρη υλοποίηση	❌
`abstract method`	Μέθοδος χωρίς σώμα — υποχρεωτική υλοποίηση	—
`final method`	Αποτρέπει override	—
`interface`	Συμβόλαιο αλληλεπίδρασης	❌
`default method`	Προαιρετική υλοποίηση σε interface (Java 8+)	—
`static method`	Βοηθητική μέθοδος σε interface (Java 8+)	—
`private method`	Εσωτερικός βοηθός σε interface (Java 9+)	—
`enum`	Κλειστό σύνολο τιμών	❌
`sealed class`	Ελεγχόμενη ιεραρχία κλάσεων (Java 17+)	❌

Πηγές

- Cay Horstmann, *Η γλώσσα προγραμματισμού JAVA — Αναλυτική Προσέγγιση*, εκδόσεις Broken Hill
- Joyce Farrell, *JAVA Εκμάθηση με πρακτικά παραδείγματα*, Εκδόσεις ΚΡΙΤΙΚΗ
- Paul Deitel & Harvey Deitel, *Java How to Program*, 10/e
- Γρηγόρης Τσουμάκας, *Αντικειμενοστρεφής Προγραμματισμός — Ενότητα 7*
- Νικόλαος Θ. Λιόλιος, *Αντικειμενοστραφής Προγραμματισμός Ι — Ενότητα 9*
- https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html
- http://docs.oracle.com/javase/tutorial/java/landl/abstract.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
- https://openjdk.org/jeps/409 *(Sealed Classes — JEP 409, Java 17)*
- https://openjdk.org/jeps/441 *(Pattern Matching for switch — JEP 441, Java 21)*