Αντικειμενοστραφής Προγραμματισμός

Β' εξάμηνο – Τμήμα Πληροφορικής – Ιόνιο Πανεπιστήμιο

Ενότητα 6

Κληρονομικότητα · Πολυμορφισμός · Γενικές κλάσεις

Δημήτρης Ρίγγας

Μηχανικός Η/Υ & Πληροφορικής, MSc, PhD

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

riggas@ionio.gr

Κληρονομικότητα

Inheritance

Κληρονομικότητα (Inheritance)

Δημιουργία νέας κλάσης **απορροφώντας μέλη** προϋπάρχουσας κλάσης και εμπλουτίζοντάς τα με νέες ή τροποποιημένες δυνατότητες.

Γιατί;

- Εξοικονόμηση χρόνου προγραμματισμού
- Αξιοποίηση δοκιμασμένου – αποσφαλματωμένου κώδικα

Πώς;

- Ξεκινάμε από μια **υπερ-κλάση** (superclass / βασική κλάση)
- Την επεκτείνουμε δημιουργώντας μια **υπο-κλάση** (subclass / παραγόμενη κλάση)
- Η υπο-κλάση έχει τις ίδιες συμπεριφορές με την υπερ-κλάση και προσθέτει **εξειδικευμένες** συμπεριφορές

Τι κληρονομείται;

Κληρονομούνται όλα τα μέλη της υπερκλάσης:

- Πεδία (fields)
- Μέθοδοι (methods)
- Εμφωλευμένες κλάσεις (nested classes)

Δεν κληρονομούνται οι κατασκευαστές (constructors)

- Αλλά μπορούν να **κληθούν** από την υποκλάση με τη `super()`

√√√ Ακόμα και τα `private` μέλη **κληρονομούνται** — απλώς δεν είναι άμεσα προσβάσιμα από την υποκλάση.

Πόσες υπερκλάσεις μπορεί να έχει μια κλάση;

Η Java υποστηρίζει μόνο **single inheritance** (μονή κληρονομικότητα):

```
MySuperClass
    └── MySubClass
```

! Αντίθετα με τη **C++** που υποστηρίζει πολλαπλή κληρονομικότητα.

- Κάθε κλάση στη Java **άρρητα κληρονομεί** την `Object` (`java.lang.Object`) αν δεν δηλώνει ρητά κάποια άλλη
- Η `Object` είναι η **μόνη κλάση** που δεν κληρονομεί καμία άλλη

Ιεραρχία:

```
Object
  └── Post
        ├── TextPost
        └── PhotoPost
```

Πώς δημιουργούμε μια υπο-κλάση

Χρήση λέξης-κλειδί **`extends`**:

```
// Δήλωση βασικής κλάσης
public class MySuperClass {
    // ...
}

// Δήλωση υπο-κλάσης
public class MySubClass extends MySuperClass {
    // κληρονομεί όλα τα (μη-private) μέλη της MySuperClass
    // και προσθέτει τα δικά της
}
```

✓ Μια κλάση μπορεί να κάνει `extends` σε **ακριβώς μία** άλλη κλάση.

Παράδειγμα — Αναρτήσεις κοινωνικού δικτύου

Αναρτήσεις σε ένα κοινωνικό δίκτυο:

- Μηνύματα κειμένου
- Φωτογραφία με λεζάντα (χωρίς άλλο κείμενο)

```
public class Post {
    String username;
    String text;
    long timestamp;
    int likes;
}

public class PhotoPost extends Post {
    String filename;
    String caption;
}
```

- PhotoPost.username / PhotoPost.timestamp / PhotoPost.text → κληρονομήθηκαν από την Post
- Post.text → κληρονομείται και από την PhotoPost — πρόβλημα σχεδιασμού! Μια φωτογραφική ανάρτηση δεν θα έπρεπε να έχει πεδίο text. *Διορθώνεται στην επόμενη διαφάνεια.*

Καλή πρακτική – Αποφυγή αντιγραφής κώδικα

Ορίζουμε **στην υπερ-κλάση** ό,τι είναι κοινό. Στις υπο-κλάσεις προσθέτουμε **επιπλέον** πεδία και μεθόδους:

```
public class Post {
    String username;
    long timestamp;
    int likes;
}

public class TextPost extends Post {
    String text;
}

public class PhotoPost extends Post {
    String filename;
    String caption;
}
```

✓ **DRY principle** (Don't Repeat Yourself): ένας κώδικας, ένα σημείο αλλαγής.

Κατασκευαστές (Constructors) & Κληρονομικότητα

Οι κατασκευαστές **δεν κληρονομούνται**, αλλά καλούνται με τη `super()`:

Περίπτωση	Συμπεριφορά
Υπερκλάση έχει default constructor	Καλείται **άρρητα** από κατασκευαστές υποκλάσης
Υπερκλάση δεν έχει default constructor	Ο κατασκευαστής υποκλάσης **οφείλει ρητά** να καλέσει `super(...)`

✓ Η κλήση `super(...)` **πρέπει να είναι η πρώτη πρόταση** εντός του κατασκευαστή υποκλάσης.

```java
public class Post {
    private String username;
    protected long timestamp;

    public Post(String username) {
        this.username = username;
        this.timestamp = System.currentTimeMillis();
    }
}

public class PhotoPost extends Post {
    private String filename;

    public PhotoPost(String username, String filename) {
        super(username);          // ← πρώτη πρόταση, υποχρεωτικά
        this.filename = filename;
    }
}
```

Προστασία μελών & Κληρονομικότητα

Τροποποιητής	Ίδια κλάση	Ίδιο package	Υποκλάση	Παντού
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
(package-private)	✅	✅	❌*	❌
`private`	✅	❌	❌	❌

*Υποκλάσεις σε **διαφορετικό** package δεν βλέπουν package-private μέλη.

√ Τα `private` μέλη **κληρονομούνται** αλλά είναι προσβάσιμα **μόνο** μέσω `public` / `protected` μεθόδων πρόσβασης.

```
public class Post {
    private int likes;                    // κληρονομείται αλλά μη ορατό
    public void increaseLikes() { this.likes++; }
    public int getLikes() { return this.likes; }
}
```

Αναφορά σε μέλη υπερ-κλάσης — `super` vs `this`

	`this.<μέλος>`	`super.<μέλος>`
Αναφέρεται σε…	μέλη της **τρέχουσας κλάσης**	μέλη της **άμεσης υπερκλάσης**
Κλήση constructor	`this(...)` — 1η εντολή	`super(...)` — 1η εντολή
Κλήση μεθόδου	`this.method()`	`super.method()`

✓ **Δεν μπορείτε** να χρησιμοποιήσετε **και** `this()` **και** `super()` στον ίδιο constructor — και οι δύο απαιτούνται ως πρώτη εντολή!

```
public class Post {
    protected void reset() {
        this.likes = 0;
        this.timestamp = System.currentTimeMillis();
    }
}

public class PhotoPost extends Post {
    public void reset(String newFilename) {
        this.filename = newFilename;
        super.reset();          // καλεί τη reset() της υπερκλάσης
    }
}
```

Υποσκέλιση μεθόδων (Method Overriding)

Επανα-υλοποίηση από μια υποκλάση μιας μεθόδου που έχει κληρονομήσει.

Κανόνες κατά την υποσκέλιση:

- ❌ Δεν αλλάζουμε **υπογραφή** ή **επιστρεφόμενο τύπο**
- ✅ Επιτρέπεται **ευρύτερο** επίπεδο πρόσβασης (π.χ. `protected` → `public`)
- ✅ Επιτρέπεται **μείωση** ή **εξάλειψη** δηλωμένων exceptions, **όχι προσθήκη**
- ✅ Χρησιμοποιούμε πάντα `@Override` — προστατεύει από λάθη υπογραφής

```
public class Post {
    public String toString() {
        return "by " + this.username + " @ " + this.timestamp
             + "\nLiked " + this.likes + " times.";
    }
}

public class TextPost extends Post {
    @Override
    public String toString() {          // σωστή υποσκέλιση
        return this.text + "\n" + super.toString();
    }
}

public class PhotoPost extends Post {
    @Override
    public String toString(String prefix) { // compile error – διαφορετική υπογραφή
        return "Photo [" + this.filename + "]\n"
             + this.caption + "\n" + super.toString();
    }
}
```

√ Αν παραλείψετε `@Override` και αλλάξετε κατά λάθος την υπογραφή, κάνει compile αλλά δεν έχετε υποσκέλιση — έχετε **υπερφόρτωση** (overloading). Λογικό σφάλμα!

Πολυμορφισμός

Polymorphism

Πολυμορφισμός (Polymorphism)

✓ Αρχή της Βιολογίας: ένας οργανισμός μπορεί να έχει πολλές μορφές ή καταστάσεις.

Στον αντικειμενοστραφή προγραμματισμό (σε συνδυασμό με κληρονομικότητα):

✓ Προγραμματίζουμε για το **γενικό** αντί του **εξειδικευμένου**.

Ένα σύνολο αντικειμένων που μοιράζονται κοινή υπερ-κλάση αντιμετωπίζεται ως **instances της υπερ-κλάσης**.

```
// Η MyWall δουλεύει με Post – δεν "ξέρει" για TextPost/PhotoPost
public class MyWall {
    private ArrayDeque<Post> posts = new ArrayDeque<>();

    public void addPost(Post p) {   // δέχεται οποιοδήποτε Post ή υποκλάση του
        this.posts.push(p);
    }

    public void showPosts() {
        for (Post p : posts) {
            System.out.println(p);  // καλεί toString() του πραγματικού τύπου
            System.out.println("==================");
        }
    }
}
```

Παράδειγμα — Τοίχος δημοσιεύσεων (Wall)

```
public class Main {
    public static void main(String[] args) {
        PhotoPost p1 = new PhotoPost("Nancy");
        p1.setPhoto("selfie.png");
        p1.setCaption("Wow....");
        p1.increaseLikes(); p1.increaseLikes(); p1.increaseLikes();

        TextPost p2 = new TextPost("George", "Here's a thought..");
        p2.increaseLikes(); p2.increaseLikes();

        MyWall wall = new MyWall();
        wall.addPost(p1);   // PhotoPost ως Post ✅
        wall.addPost(p2);   // TextPost ως Post ✅

        wall.showPosts();
    }
}
```

✓ Η `showPosts()` γράφτηκε **μία φορά** για `Post`, αλλά λειτουργεί σωστά για **κάθε** υποκλάση — τώρα και στο μέλλον, χωρίς αλλαγή.

Εικονική κλήση μεθόδου (Virtual Method Invocation)

Φάση	Τι γίνεται
Μεταγλώττιση	Ο compiler ελέγχει αν η μέθοδος υπάρχει στην **δηλωμένη** κλάση
Εκτέλεση	Η JVM καλεί τη μέθοδο της κλάσης στην οποία **πραγματικά ανήκει** το αντικείμενο

```
Post p = new TextPost("George", "Hello!");
System.out.println(p);
// ↑ Compiler: Post έχει toString()
// ↑ JVM: καλεί TextPost.toString() – εικονική κλήση
```

Σημαντικό:

- Αν η υπερκλάση έχει μέθοδο με **ίδιο όνομα αλλά διαφορετική λίστα παραμέτρων** → δεν είναι υποσκέλιση, είναι **υπερφόρτωση** (overloading)
- Για να χρησιμοποιήσετε υποσκελισμένη μέθοδο: `super.methodName()`
- Σε αντίθεση με constructors:
  - η `super.method()` **δεν χρειάζεται** να είναι πρώτη εντολή και
  - **δεν καλείται αυτόματα**

Προσοχή! StackOverflowError

```java
public class TextPost extends Post {
    @Override
    public String toString() {
        return this.text + "\n"
             + this.toString();   // 💥 ΛΑΘΟΣ – αναδρομική κλήση στον εαυτό της!
    }
}
```

```
Exception in thread "main" java.lang.StackOverflowError
    at TextPost.toString(TextPost.java:10)
    at TextPost.toString(TextPost.java:10)
    at TextPost.toString(TextPost.java:10)
    ...
```

Η κλήση `this.toString()` (ή χωρίς qualifier) μέσα σε υποσκελισμένη μέθοδο **καλεί την ίδια μέθοδο αναδρομικά**.
Χρησιμοποιήστε `super.toString()`.

Επιτρεπόμενες Εκχωρήσεις (Assignments)

```
// ✅ Αναφορά υπερκλάσης → αντικείμενο υπερκλάσης
Post p3 = new Post("George");

// ✅ Αναφορά υποκλάσης → αντικείμενο υποκλάσης
TextPost p4 = new TextPost("Nancy", "Testing post..");

// ✅ Αναφορά υπερκλάσης → αντικείμενο υποκλάσης
//   (πρόσβαση μόνο σε μέλη υπερκλάσης χωρίς cast)
Post p5 = new PhotoPost("George");
// p5.setPhoto("photo.jpg");           // ❌ COMPILER ERROR

((PhotoPost) p5).setPhoto("photo.jpg"); // ✅ explicit casting

// ❌ Αναφορά υποκλάσης → αντικείμενο υπερκλάσης: ΔΕΝ επιτρέπεται
// PhotoPost p6 = new Post("Nancy");   // ❌ COMPILER ERROR
```

√ Η κληρονομικότητα είναι **μονόδρομη**: υποκλάση IS-A υπερκλάση, όχι το ανάποδο.

Η κλάση Object

java.lang.Object

Ιεραρχία κλάσεων στη Java

Κάθε κλάση που δεν κάνει ρητά `extends` κάποιας άλλης, **άρρητα κληρονομεί** την `Object`.

Βασικές μέθοδοι της `Object`:

Μέθοδος	Περιγραφή
`public String toString()`	Αναπαράσταση αντικειμένου ως String
`public boolean equals(Object obj)`	Σύγκριση αντικειμένων (λογική ισότητα)
`public int hashCode()`	Αριθμητική κωδικοποίηση (για Maps, Sets)
`public final Class<?> getClass()`	Επιστρέφει την κλάση του αντικειμένου
`protected Object clone()`	Ρηχό αντίγραφο αντικειμένου
`public final void wait() / notify()`	Συγχρονισμός νημάτων

✓ Οι υποκλάσεις συνήθως **υποσκελίζουν** `toString()`, `equals()` και `hashCode()`.

✓ ⚠️ Αν υποσκελίσετε `equals()`, **οφείλετε** να υποσκελίσετε και `hashCode()` — διαφορετικά σπάτε το συμβόλαιο των collections (HashMap, HashSet).

Τελεστής `instanceof`

Προσδιορισμός τύπου αντικειμένου **κατά το χρόνο εκτέλεσης**:

```
// Σύνταξη: αναφορά instanceof Κλάση
if (p instanceof TextPost)
    texts++;
else if (p instanceof PhotoPost)
    photos++;
```

“ ⚠️ **Κακή πρακτική** όταν η λογική εξαρτάται από πλήθος υποκλάσεων:

- Τι θα συμβεί αν προστεθεί νέα υποκλάση `VideoPost`;
- Ο κώδικας σπάει σε πολλά σημεία. ”

Προτιμήστε **πολυμορφισμό** (μέθοδοι στις ίδιες τις κλάσεις) όπου είναι δυνατό.

Μέθοδος `getClass()` – εναλλακτική του `instanceof`

```
import java.util.HashMap;

public class MyWall {
    private HashMap<String, Integer> countPostsByClass = new HashMap<>();

    public void addPost(Post p) {
        this.posts.push(p);
        String className = p.getClass().getName();
        Integer prevCount = countPostsByClass.get(className);
        countPostsByClass.put(className, prevCount == null ? 1 : prevCount + 1);
    }

    public void showPosts() {
        System.out.println("Total posts: " + posts.size()
            + " " + countPostsByClass);
        for (Post p : posts) {
            System.out.println(p);
            System.out.println("==================");
        }
    }
}
```

✓ **Πλεονέκτημα:** Λειτουργεί αυτόματα για οποιαδήποτε νέα υποκλάση χωρίς αλλαγή κώδικα.

Pattern Matching για `instanceof` Java 16+ 🆕

Από την **Java 16** (preview: Java 14–15), το `instanceof` υποστηρίζει **pattern matching**:

Παλαιός τρόπος (Java ≤ 15):

```
if (p instanceof TextPost) {
    TextPost tp = (TextPost) p;   // χειροκίνητο cast – verbose!
    System.out.println(tp.getText());
}
```

Νέος τρόπος με Pattern Matching (Java 16+):

```
if (p instanceof TextPost tp) {   // cast + binding σε μία εντολή
    System.out.println(tp.getText());
}
```

Ακόμα πιο εκφραστικό με `switch` (Java 21+):

```
String result = switch (p) {
    case TextPost tp  -> "Text: " + tp.getText();
    case PhotoPost pp -> "Photo: " + pp.getFilename();
    default           -> "Unknown post type";
};
```

✓ Οι **binding variables** (`tp`, `pp`) είναι διαθέσιμες **μόνο** στο scope όπου ισχύει η συνθήκη.

Sealed Classes — Ελεγχόμενη Ιεραρχία Java 17+ 🆕

Από την **Java 17**, οι **sealed classes** επιτρέπουν να ορίσετε **ακριβώς ποιες κλάσεις** μπορούν να επεκτείνουν μια κλάση (ή να υλοποιούν ένα interface):

```
// Μόνο TextPost, PhotoPost και VideoPost επιτρέπεται να επεκτείνουν την Post
public sealed class Post permits TextPost, PhotoPost, VideoPost {
    // ...
}

// Κάθε υποκλάση πρέπει να δηλώσει αν είναι:
public final class TextPost extends Post { ... }          // δεν επεκτείνεται περαιτέρω
public non-sealed class PhotoPost extends Post { ... }  // μπορεί να επεκταθεί ελεύθερα
public sealed class VideoPost extends Post               // επεκτείνεται μόνο από:
    permits ShortVideo, LongVideo { ... }
```

Γιατί;

- Ο compiler **γνωρίζει όλες τις υποκλάσεις** → exhaustive `switch` χωρίς `default`
- Ασφαλέστερος σχεδιασμός API — κανείς δεν μπορεί να προσθέσει "έκπληξη" υποκλάση
- Συνεργάζεται άριστα με pattern matching

```
// Ο compiler ξέρει ότι Post είναι πάντα TextPost, PhotoPost ή VideoPost
String label = switch (p) {
    case TextPost  tp -> "📝 " + tp.getText();
    case PhotoPost pp -> "📷 " + pp.getFilename();
    case VideoPost vp -> "🎬 " + vp.getDuration() + "s";
    // Δεν χρειάζεται default — exhaustive!
};
```

Γενικές Κλάσεις

Generics — Μέρος Α'

Γενικές κλάσεις (Generics)

Δομές οι οποίες μπορούν να χρησιμοποιηθούν σε συνδυασμό με **πολλούς τύπους**:

```
ArrayList<String>   // λίστα Strings
ArrayList<Order>    // λίστα Orders
HashMap<String, Integer>  // map από String σε Integer
```

Ορισμός:

- Κάθε γενική κλάση έχει έναν ή περισσότερους **παραμετρικούς τύπους** (type parameters)
- Οι παραμετρικοί τύποι χρησιμοποιούνται στις μεθόδους για δήλωση ορισμάτων / επιστρεφόμενων τιμών

Σύμβαση ονοματολογίας:

Γράμμα	Σημασία
`E`	Element (στοιχείο συλλογής)
`K`	Key (κλειδί map)
`V`	Value (τιμή map)
`T`, `S`, `U`, `V`	οποιοσδήποτε τύπος

✓ Οι παραμετρικοί τύποι μπορούν να είναι **μόνο αναφερόμενοι** (reference) τύποι, **όχι πρωτεύοντες** (int, double, ...).

Υλοποίηση μιας γενικής κλάσης

```
public class Pair<F, S> {
    private F first;
    private S second;

    public Pair(F f, S s) {
        this.first  = f;
        this.second = s;
    }

    public F getFirst()  { return this.first; }
    public S getSecond() { return this.second; }

    @Override
    public String toString() {
        return this.first + " - " + this.second;
    }
}
```

- Οι παραμετρικοί τύποι `<F, S>` δηλώνονται μετά το όνομα της κλάσης
- Χρησιμοποιούνται **ως τύποι** πεδίων, παραμέτρων, επιστρεφόμενων τιμών
- Μπορεί να υπάρχουν και **μη-παραμετρικά** μέλη

Αξιοποίηση μιας γενικής κλάσης

```
public class Main {
    public static void main(String[] args) {
        Pair<String, Integer>  p1 = new Pair<>("A", 1);
        System.out.println(p1);   // A - 1

        Pair<Integer, Integer> p2 = new Pair<>(2, 1);
        System.out.println(p2);   // 2 - 1

        Pair<Integer, String>  p3 = new Pair<>(3, "C");
        System.out.println(p3);   // 3 - C
    }
}
```

- Οι πραγματικοί τύποι δηλώνονται εντός `<>` κατά τη δήλωση αντικειμένου
- Η σειρά αντιστοιχεί στη σειρά των παραμετρικών τύπων
- Το diamond operator `<>` (Java 7+): ο compiler συμπεράνει τους τύπους από τη δήλωση

Γενικές μέθοδοι (Generic Methods)

Μέθοδοι με παραμετρικούς τύπους — μπορούν να ορίζονται και σε **μη-γενικές** κλάσεις:

```
import java.util.Collection;

public class CollectionUtil {           // απλή (μη-γενική) κλάση

    public static <E> String toString(Collection<E> c) {  // γενική μέθοδος
        StringBuilder sb = new StringBuilder();
        for (E e : c) {
            sb.append(e.toString()).append(", ");
        }
        sb.replace(sb.length() - 2, sb.length() - 1, "");
        return sb.toString().trim();
    }
}
```

- Ο παραμετρικός τύπος `<E>` δηλώνεται **μεταξύ** του access modifier και του επιστρεφόμενου τύπου
- Κατά την κλήση **δεν** δηλώνουμε τον τύπο — ο compiler τον **συμπεραίνει** από τα ορίσματα

```
CollectionUtil.toString(listOfStrings);  // E συμπεραίνεται ως String
CollectionUtil.toString(listOfPairs);    // E συμπεραίνεται ως Pair<...>
```

Αξιοποίηση γενικής μεθόδου

```
ArrayList<Pair<String, Integer>> pairs = new ArrayList<>();
pairs.add(new Pair<>("A", 1));
pairs.add(new Pair<>("B", 2));
pairs.add(new Pair<>("C", 3));
System.out.println(CollectionUtil.toString(pairs));
// Output: A - 1, B - 2, C - 3

ArrayList<Pair<String, Double>> pairs2 = new ArrayList<>();
pairs2.add(new Pair<>("A", 0.1));
pairs2.add(new Pair<>("B", 0.2));
pairs2.add(new Pair<>("C", 0.3));
System.out.println(CollectionUtil.toString(pairs2));
// Output: A - 0.1, B - 0.2, C - 0.3
```

✓ Διαφορά από γενική κλάση: Στη γενική κλάση δηλώνουμε `new Pair<String, Integer>(...)`, στη γενική μέθοδο απλώς `CollectionUtil.toString(...)` — ο compiler βρίσκει τον τύπο μόνος του.

Bounded Type Parameters — Ορισμένοι Παραμετρικοί Τύποι

Μπορούμε να **περιορίσουμε** τους τύπους που επιτρέπονται ως παραμετρικοί:

```
// Μόνο τύποι που υλοποιούν Comparable<T>
public class SortedPair<T extends Comparable<T>> {
    private T smaller, larger;

    public SortedPair(T a, T b) {
        if (a.compareTo(b) <= 0) { smaller = a; larger = b; }
        else                     { smaller = b; larger = a; }
    }
    public T getSmaller() { return smaller; }
    public T getLarger()  { return larger;  }
}

SortedPair<Integer> sp = new SortedPair<>(5, 3);  // ✅
SortedPair<String>  ss = new SortedPair<>("z","a"); // ✅
// SortedPair<Post> sp2 = ...;  // ❌ αν Post δεν υλοποιεί Comparable
```

Συντακτικό:

- `<T extends Τύπος>` — T πρέπει να είναι υποκλάση ή να υλοποιεί τον Τύπο
- `<T extends A & B & C>` — T πρέπει να υλοποιεί **πολλαπλά** interfaces (πολλαπλά bounds)

Type Erasure — Διαγραφή Τύπων

Τα generics στη Java υλοποιούνται μέσω **type erasure**: οι παραμετρικοί τύποι **αφαιρούνται κατά τη μεταγλώττιση** και αντικαθίστανται από τον bound τους (ή `Object`).

```
// Κώδικας που γράφουμε:
List<String> strings = new ArrayList<>();
strings.add("hello");
String s = strings.get(0);

// Τι βλέπει η JVM (μετά από erasure):
List strings = new ArrayList();
strings.add("hello");
String s = (String) strings.get(0);  // ← ο compiler εισάγει cast αυτόματα
```

Συνέπειες:

- `new T()` — ❌ αδύνατο (δεν ξέρουμε τον τύπο T κατά την εκτέλεση)
- `instanceof List<String>` — ❌ αδύνατο (ο compiler δεν γνωρίζει)
- `instanceof List<?>` — ✅ επιτρέπεται
- Κατά την **εκτέλεση** τας `ArrayList<String>` και `ArrayList<Integer>` είναι **ίδιος τύπος**!

✓ Ο λόγος είναι **συμβατότητα με παλαιότερο κώδικα** (pre-Java 5).

Πηγές

- Cay Horstmann, *Η γλώσσα προγραμματισμού Java — Αναλυτική Προσέγγιση*, Broken Hill
- Joyce Farrell, *Java — Εκμάθηση με πρακτικά παραδείγματα*, Εκδόσεις Κριτική
- Paul & Harvey Deitel, *Java How to Program, 10/e*
- Γρηγόρης Τσουμάκας, *Αντικειμενοστρεφής Προγραμματισμός — Ενότητα 7*
- Νικόλαος Θ. Λιόλιος, *Αντικειμενοστρεφής Προγραμματισμός Ι — Ενότητα 9*
- https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html
- https://docs.oracle.com/javase/tutorial/java/landl/polymorphism.html
- https://docs.oracle.com/en/java/javase/21/language/pattern-matching.html *(νέο)*
- https://openjdk.org/jeps/409 — Sealed Classes (JEP 409, Java 17) *(νέο)*
- https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html *(νέο)*