ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS
ΚΕΝΤΡΟ ΕΠΙΜΟΡΦΩΣΗΣ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗΣ

Nested classes
Anonymous classes & Lambdas, callbacks
Mixin interfaces

Αθ. Ανδρούτσος

Nested Classes

- Μία nested class είναι μία κλάση που ορίζεται μέσα σε μία άλλη κλάση
- Μία nested class υπάρχει μόνο για να εξυπηρετεί την outer class
- Υπάρχουν τέσσερις τύποι nested classes, 1) static, 2) non-static, 3) anonymous, 4) local

Static nested classes (1)

- Μία static member class είναι η πιο απλή μορφή nested class
- Είναι σαν να έχουμε μία κανονική κλάση, απλά να την έχουμε ορίσει μέσα σε μια άλλη κλάση (enclosing class) ώστε να έχει πρόσβαση σε όλα τα static μέλη της enclosing class ακόμα και στα private
- Ακολουθεί τους κανόνες των static class members και αν δηλωθεί private είναι accessible μόνο μέσα από την enclosing κλάση

Static nested classes (2)

- Μία κοινή χρήση των static nested κλάσεων είναι σαν *public helper classes* που λειτουργούν συνεργατικά με την enclosing class
- Για παράδειγμα μπορεί να έχω ένα Operation enum που να είναι public static member μιας enclosing class ώστε να αναφέρομαι ως Calculator.Operation.PLUS ή Calculator.Operation.MINUS

Non-static nested class (1)

- Τεχνικά οι static και non-static member classes μπορεί να διαφέρουν μόνο ως προς το keyword static, αλλά στην πράξη πρόκειται για τελείως διαφορετικές κλάσεις μιας και οι non-static nested classes είναι instance members που δημιουργούνται μαζί με ένα enclosing instance της outer class

Non-static nested class (2)

- Έχουν πρόσβαση σε όλα τα μέλη της outer class και επίσης μπορούν να αναφερθούν στο enclosing instance μέσω του this, άρα το scope τους είναι η outer class
- Δημιουργούνται κάνοντας new τον constructor τους από τον constructor ή μέθοδο της outer class

Non-static nested class (3)

- Μία κοινή χρήση non-static member classes είναι το *Adapter Design Pattern* όπου επιτρέπει μία εξωτερική κλάση να επιστρέφει ένα instance μία άλλης unrelated class
- Όπως για παράδειγμα όπως για παράδειγμα αν μέσα από μία κλάση Theater μπορούμε να επιστρέφουμε τις θέσεις του Θεάτρου μέσα από ένα method getSeatCollection() που επιστρέφει ένα collection από Seat

Builder Design Pattern (1)

- Το πρόβλημα που επιλύει το Builder Design pattern είναι το εξής
- Έστω ότι έχουμε μία κλάση με πολλά πεδία, έστω τρία. Αν έχουμε τρία πεδία προαιρετικά τότε για ευκολία των προγραμματιστών θα πρέπει να παρέχουμε οκτώ δημιουργούς (έναν default, τρείς με ένα πεδίο, τρεις με δύο πεδία, και ένα και με τα τρία πεδία) (telescoping constructor pattern)

Builder Design Pattern (2)

- Αν δημιουργήσουμε ένα default constructor και αρχικοποιήσουμε με setters θα υπάρχει ασυνέπεια του αντικειμένου μεταξύ των κλήσεων στους setters
- Ενώ επίσης το JavaBeans pattern αποκλείει τη δυνατότητα να είναι η κλάση immutable γιατί κανονικά το JavaBeans pattern θέλει και setters

Builder Design Pattern (3)

- Ευτυχώς υπάρχει και μία ακόμα εναλλακτική που συνδυάζει το safety και convenience του telescoping pattern με το readability του JavaBeans pattern
- Το **Builder Design Pattern**, όπου η κλάση μας μπορεί να είναι immutable και να περιλαμβάνει μία inner static Builder class με τα ίδια πεδία
- Η Builder class έχει κανονικά constructor για τα final πεδία και ειδικούς setters για τα μη-final πεδία. Επίσης έχει μία μέθοδο build που επιστρέφει μέσω του private constructor της κλάσης μας ένα instance της κλάσης μας

Book builder pattern (4)


```java
package testbed.ch17.marker;

public class Book implements Item {
    private final long id;
    private final String isbn;
    private final String author;
    private final String title;

    public static class Builder {
        // Required parameters
        private final long id;
        private final String isbn;

        // Optional parameters - initialized to default values
        private String author   = "";
        private String title    = "";
```

- Η κλάση book περιέχει μία inner public static Builder κλάση
- Η κλάση Book είναι immutable
- Η Builder class έχει κάποια πεδία immutable και κάποια mutable που μπορούν να γίνουν set

Book builder pattern (5)


```
18          // Could be a static factory
19          public Builder(long id, String isbn) {
20              this.id     = id;
21              this.isbn   = isbn;
22          }
23
24          public Builder author(String author) {
25              this.author = author;
26              return this;
27          }
28
29          public Builder title(String title) {
30              this.title = title;
31              return this;
32          }
33
34          public Book build() {
35              return new Book(this);
36          }
37      }
```

- Ο constructor της Builder αρχικοποιεί τα required πεδία
- Έχουμε επίσης τρεις ειδικούς setters που επιστρέφουν το this

Book builder pattern (6)


```
39    @     private Book(Builder builder) {
40              this.id     = builder.id;
41              this.author = builder.author;
42              this.isbn   = builder.isbn;
43              this.title  = builder.title;
44          }
45
46          public long getId() { return id; }
49          public String getIsbn() { return isbn; }
52          public String getAuthor() { return author; }
55          public String getTitle() { return title; }
58
59          @Override
60          public String toString() {...}
68      }
```

- Ο constructor της Book είναι private και παίρνει ως παράμετρο το builder instance

Main (1)


```java
package testbed.ch17.marker;

public class Main {

    public static void main(String[] args) {
        Book book1 = new Book.Builder(1, "123").build();
        Book book2 = new Book.Builder(2, "345")
                .author("Th.")
                .title("Java I")
                .build();
    }

```

- Με το builder design pattern δημιουργούμε δύο book instances

Anonymous classes (1)

- Πρόκειται για κλάσεις χωρίς όνομα. Δεν είναι μέλη κάποιας outer class
- Ταυτόχρονα δηλώνονται και γίνονται instantiate με new στο σημείο της δήλωσης
- Επιτρέπεται να δηλώνουμε ανώνυμες κλάσεις σε οποιοδήποτε σημείο του κώδικα θα μπορούσαμε να έχουμε ένα κανονικό named instance της κλάσης

Anonymous classes (2)

- Ανώνυμες κλάσεις χρησιμοποιούμε όταν χρειάζεται να κάνουμε invoke ένα instance της κλάσης μόνο μία φορά
- Επειδή οι ανώνυμες κλάσεις είναι expressions θα πρέπει το σώμα τους να παραμένει μικρό, δέκα γραμμές κώδικα ή και λιγότερο
- Πριν από τα lambdas οι ανώνυμες κλάσεις χρησιμοποιούνταν για να υλοποιούμε callback functions (υλοποιώντας functional interfaces) αλλά τώρα τα lambdas είναι προτιμότερα

Ανώνυμες (inner) κλάσεις (3)

- Όταν η ανώνυμη κλάση αφορά υλοποίηση interface θα πρέπει ταυτόχρονα με την δημιουργία της να ορίζει και τις μεθόδους του Interface
- Όταν το interface είναι Functional τότε αντί για ανώνυμη κλάση μπορούμε να χρησιμοποιήσουμε Lambda expressions ή Method Reference

Local classes

- Οι local classes είναι ο λιγότερο χρησιμοποιούμενος τύπος nested class
- Ορίζονται οπουδήποτε μπορούμε να ορίσουμε local variables, όπως μέσα σε μία μέθοδο
- Μπορεί να έχουν όνομα ή να είναι ανώνυμες αν υλοποιούν ένα interface

Callback μέθοδοι (1)

- Υπάρχουν περιπτώσεις που θα θέλαμε να περνάμε σε μία μέθοδο ως παράμετρο άλλες μεθόδους, ώστε να μπορεί η αρχική μέθοδος να τις καλεί, παρέχοντας έτσι ευελιξία σε χρόνο εκτέλεσης σχετικά με ποια μέθοδο θα καλέσει
- Η μέθοδος που περνάει ως παράμετρος ονομάζεται **callback**

Callback μέθοδοι (2)

- Αν θα θέλαμε για παράδειγμα να εφαρμόσουμε ένα συγκεκριμένο αλγόριθμο για την ταξινόμηση ενός πίνακα σε μία μέθοδο που ταξινομεί πίνακες θα μπορούσαμε να έχουμε κάποιους διαθέσιμους αλγόριθμους και να περνάμε ως παράμετρο το όνομα εκείνης της μεθόδου που υλοποιεί τον αλγόριθμο που θέλουμε κάθε φορά

Callback μέθοδοι (3)

- Τα callbacks είναι χρήσιμα στο **event-driven programming**, γιατί μπορεί μία μέθοδος να 'ακούει' (listen) για ένα event κι όταν αυτό συμβεί να εκτελεστεί ένα callback
- Τότε θα είχαμε μια μορφή π.χ. btn.addListener('event', callback() {})
- Αν το event ήταν το completion ενός async task, τότε το callback θα ήταν τι να κάνουμε onSuccess και τι onError, οπότε μπορούμε με callbacks να υλοποιήσουμε και **ασύγχρονο προγραμματισμό**

Callback μέθοδοι (3)

- Επειδή η Java δεν έχει συναρτήσεις μέχρι την Java 7 ως callback περνούσαμε ανώνυμες κλάσεις που υλοποιούσαν functional interfaces
- Από την **Java 8 και μετά ο προτιμώμενος τρόπος είναι να χρησιμοποιούμε lambdas ή και method references**

Παράδειγμα

- Έστω ότι έχουμε το παράδειγμα ενός Ιππότη που αναλαμβάνει αποστολές
- Αν έχουμε λοιπόν για τον Ιππότη μία μέθοδο *embarkOnMission(IMission)* όπου **IMission είναι Functional interface**, μπορούμε να κάνουμε inject "ανώνυμες αποστολές" στον Ιππότη ως ουσιαστικά callback functions
- Βλέπουμε δηλαδή τα functional interfaces ως function objects

interfaces


```java
package testbed.ch17.knight;

@FunctionalInterface
public interface IKnight {

    void embarkOnMission(IMission mission);
}
```

```java
package testbed.ch17.knight;

@FunctionalInterface
public interface IMission {

    void embark();
}
```

- Ορίζουμε τα interfaces *IKnight* και κυρίως I*Mission* που περνάει ως παράμετρος στο *embarkOnMission*

Knight class


```java
package testbed.ch17.knight;

public class Knight implements IKnight {

    @Override
    public void embarkOnMission(IMission mission) {
        mission.embark();
    }
}
```

- Στη θέση του IMission θα περάσει ένα callback, δηλαδή ένα lambda μιας και το IMission είναι functional interface

Main - Ανώνυμη κλάση ως callback


```
1  package testbed.ch17.knight;
2
3  public class PlayTheGame {
4
5      public static void main(String[] args) {
6
7          IKnight joa = new Knight();
8          IKnight saintGeorge = new Knight();
9
10         joa.embarkOnMission(new IMission() {
11             @Override
12             public void embark() {
13                 System.out.println("Save the princess");
14             }
15         });
16
17         saintGeorge.embarkOnMission(new IMission() {
18             @Override
19             public void embark() {
20                 killTheDragon();
21             }
22         });
23     }
25     public static void killTheDragon() {
26         System.out.println("Kill the Dragon");
27     }
28 }
```

- Παρατηρούμε πως έχουμε μία ανώνυμη κλάση για την αποστολή της *joa*
- Καθώς και μία ανώνυμη κλάση για την αποστολή του *saintGeorge*, μόνο που εδώ **χρησιμοποιούμε μία ήδη υπάρχουσα μέθοδο**, την *killTheDragon*()

Lambda Expressions


```
1   package testbed.ch17.knight;
2
3   public class PlayTheGame {
4
5       public static void main(String[] args) {
6
7           IKnight joa = new Knight();
8           IKnight saintGeorge = new Knight();
9
10          joa.embarkOnMission( () -> System.out.println("Save the princess"));
11          saintGeorge.embarkOnMission(() -> killTheDragon());
12      }
```

- Αντί για ανώνυμες κλάσεις, όταν έχουμε Functional interfaces, προτιμούμε Lambdas, όπως στις γραμμές 10, 11

Method References


```
1  package testbed.ch17.knight;
2
3  public class PlayTheGame {
4
5      public static void main(String[] args) {
6
7          IKnight joa = new Knight();
8          IKnight saintGeorge = new Knight();
9
10         // Method References
11         joa.embarkOnMission(PlayTheGame::saveThePrincess);
12         saintGeorge.embarkOnMission(PlayTheGame::killTheDragon);
13
14     }
15
16     public static void killTheDragon() {
17         System.out.println("Kill the Dragon");
18     }
19
20     public static void saveThePrincess() {
21         System.out.println("Save the princess");
22     }
23 }
```

- Αν το callback χρησιμοποιεί μία **ήδη υπάρχουσα μέθοδο**, τότε αντί για lambda μπορούμε να καλέσουμε Method Reference με τον **τελεστή ::** (γραμμές 11, 12)

Mixin interfaces

- ***Runnable*** – πρόκειται για functional interface που ορίζει τη μέθοδο run. Χρησιμοποιείται όταν δημιουργούμε threads.
- ***Cloneable*** – πρόκειται για marker interface που υποστηρίζει την αντιγραφή αντικειμένων
- ***Serializable*** – marker interface που υποστηρίζει τη μετατροπή των πεδίων μιας κλάσης σε byte-stream και την αντιγραφή τους σε αρχεία .ser

Threads (1)

- Τα threads (νήματα) είναι ανεξάρτητα προγράμματα που καταναλώνουν **χρόνο επεξεργαστή και μνήμη**
- Ωστόσο τα threads (εν αντιθέσει με τα processes) μέσα σε μια διεργασία (process) χρησιμοποιούν την ίδια **κοινή μνήμη**
- Για το λόγο αυτό χρειάζεται προσοχή ο **συγχρονισμός (synchronized)**, ποιο thread γράφει και πότε, ώστε στη συνέχεια το άλλο thread να διαβάσει σωστά, δηλαδή μετά την εγγραφή

Threads (2)

- Διαφορετικά αν δεν υπάρχει συγχρονισμός δημιουργούνται **race conditions** (ανταγωνίζονται τα threads ποιο θα πάρει πρώτο χρόνο επεξεργαστή)
- Έτσι αν την ίδια στιγμή που ένα thread πάει να γράψει σε ένα πίνακα για παράδειγμα, ένα άλλο thread προλάβει να διαβάσει, θα διαβάσει μια τιμή που δεν είναι έγκυρη, αφού στη συνέχεια η τιμή αλλάζει από το αρχικό thread

Threads (3)

- Γιαυτό το λόγο τα συστήματα που χρησιμοποιούν threads θα πρέπει να είναι thread-safe δηλαδή να κάνουν lock τη δομή που επεξεργάζονται ώστε να μην υπάρχει περίπτωση ένα άλλο thread ταυτόχρονα να επεξεργαστεί την ίδια δομή
- Το locking γενικά επιτυγχάνεται με **σηματοφορείς** που στην Java υλοποιούνται με τις wait και notify που λειτουργούν όπως τα φανάρια που ρυθμίζουν την κυκλοφορία, δηλαδή το κόκκινο σημαίνει περίμενε (wait) και το πράσινο προχώρα (notify)

Monitors

- Η Java εκτός από σηματοφορείς μας παρέχει και ένα πιο υψηλού επιπέδου μηχανισμό για locking ώστε να μπορούμε να συγχρονίζουμε threads
- Οι **monitors** είναι οι κοινές δομές μνήμης που χρησιμοποιούν τα threads
- Τους monitors τους αναθέτουμε σε objects, arrays, collections και γενικά σε δομές δεδομένων και τους βάζουμε μέσα σε synchronized blocks

Critical section

- Μέσα στα synchronized blocks βάζουμε το **critical section του κώδικα**, το μέρος του κώδικα δηλαδή που επεξεργάζεται την κοινή δομή μνήμης
- Τυπικά το critical section θα πρέπει σίγουρα να περιέχει τα writes/deletes/updates.
- Αν ο κώδικάς μας δεν κάνει write/delete/update στη κοινή δομή ή αν η κοινή δομή είναι immutable (π.χ. String) δεν χρειαζόμαστε synchronization

Runnable

- Το *functional interface* **Runnable** παρέχει τη μέθοδο **run()** και χρησιμοποιείται για τη δημιουργία threads, δηλαδή κλάσεων που κάνουν implement το Runnable και override την run().
- Πρόκειται για παρόμοιο μηχανισμό με την **κλάση Thread** την οποία μπορούμε να κάνουμε extend και να κάνουμε override την run
- Με το interface Runnable έχουμε μεγαλύτερη ευελιξία σε σχέση με την κλάση Thread στο να μπορούμε να κάνουμε και extends μία κλάση και implements το Runnable

Start a thread

- Επομένως είτε κάνουμε subclass την κλάση Thread ή (προτιμότερο) δημιουργούμε ένα Runnable object
- Και στις δύο περιπτώσεις ξεκινούμε το thread με thread.start()
- (new HelloThread()).start
- (new Thread(new HelloRunnbale()).start()

Παράδειγμα

- Έστω ότι έχουμε μία σειρά από αποστολές (ένα πίνακα missions από IMission) που αναλαμβάνει ένας Ιππότης
- Κάθε Ιππότης τρέχει σε ένα thread και μπορεί να αναλάβει μία αποστολή, την πρώτη που είναι διαθέσιμη
- Θα ελέγχει δηλαδή τον πίνακα missions και στο 1ο mission που θα βρει με mission status NOT_STARTED θα αναλαμβάνει την αποστολή, και θα κάνει το status STARTED

IKnight, IMission


```java
package gr.aueb.cf.ch17.runna;

@FunctionalInterface
public interface IKnight {

    IMission[] missions = {
            new KillTheDragonMission(),
            new SaveThePrincessMission(),
            new KillTheDragonMission()
    };

    void embarkOnMission(IMission mission);
}
```

```java
package gr.aueb.cf.ch17.runna;

public enum MissionStatus {
    STARTED,
    NOT_STARTED
}
```

```java
package gr.aueb.cf.ch17.runna;

public interface IMission {
    MissionStatus getStatus();
    void setStatus(MissionStatus missionStatus);
    void embark();
}
```

- Έχουμε ένα *enum* με mission status
- Στο *IKnight* έχουμε ένα static final πίνακα από missions

Kill the dragon mission


```java
package gr.aueb.cf.ch17.runna;

public class KillTheDragonMission implements IMission {
    private MissionStatus missionStatus = MissionStatus.NOT_STARTED;

    @Override
    public MissionStatus getStatus() {
        return missionStatus;
    }

    @Override
    public void setStatus(MissionStatus missionStatus) {
        this.missionStatus = missionStatus;
    }

    @Override
    public void embark() {
        System.out.println("Kill the dragon mission");
    }
}
```

- Kill the Dragon Mission με status = not started

Save the princess mission


```java
package gr.aueb.cf.ch17.runna;

public class SaveThePrincessMission implements IMission {
    private MissionStatus missionStatus = MissionStatus.NOT_STARTED;

    @Override
    public MissionStatus getStatus() {
        return missionStatus;
    }

    @Override
    public void setStatus(MissionStatus missionStatus) {
        this.missionStatus = missionStatus;
    }

    @Override
    public void embark() {
        System.out.println("Save the princess mission");
    }
}
```

- Save the princess mission με status = not started

Υλοποίηση Knight ως Thread (1)


```
1  package gr.aueb.cf.ch17.runna;
2
3  public class Knight implements IKnight, Runnable {
4      private final String name;
5
6      public Knight(String name) { this.name = name; }
9      public String getName() { return name; }
12
13     @Override
14     public void embarkOnMission(IMission mission) {
15         System.out.println(this.getName());
16         mission.setStatus(MissionStatus.STARTED);
17         mission.embark();
18     }
```

- Ο knight είναι *Runnabe* δηλαδή μπορεί να εκχωρηθεί σε ένα thread
- Η embark on mission εμφανίζει το όνομα του instance, κάνει το status STARTED, και μετά κάνει embark

Υλοποίηση Knight ως Thread (2)

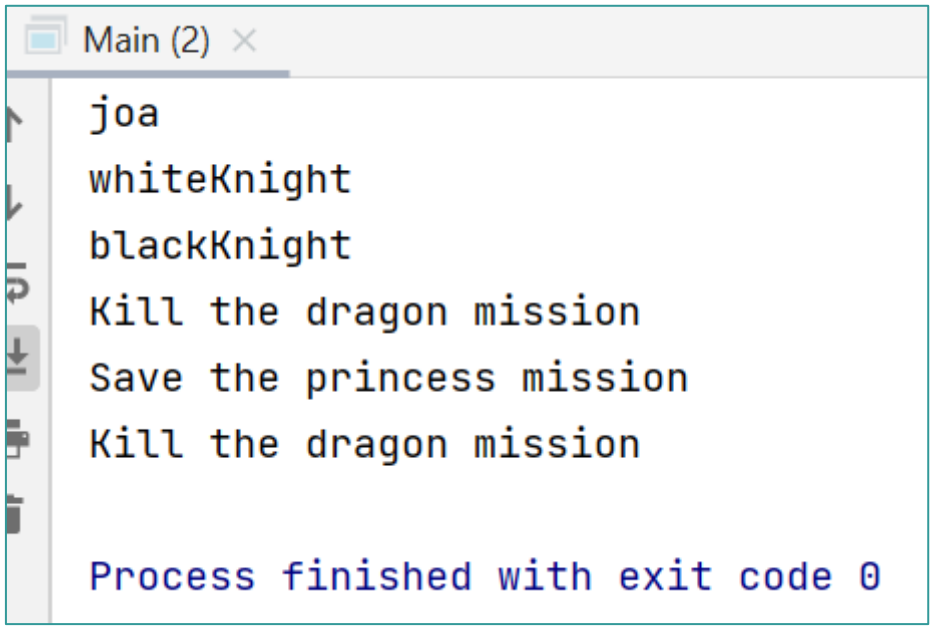

```
20      @Override
21      public void run() {
22          synchronized (IKnight.missions) {
23              for (IMission mission : IKnight.missions) {
24                  if (mission.getStatus() == MissionStatus.NOT_STARTED) {
25                      this.embarkOnMission(mission);
26                      break;
27                  }
28              }
29          }
30      }
31  }
```

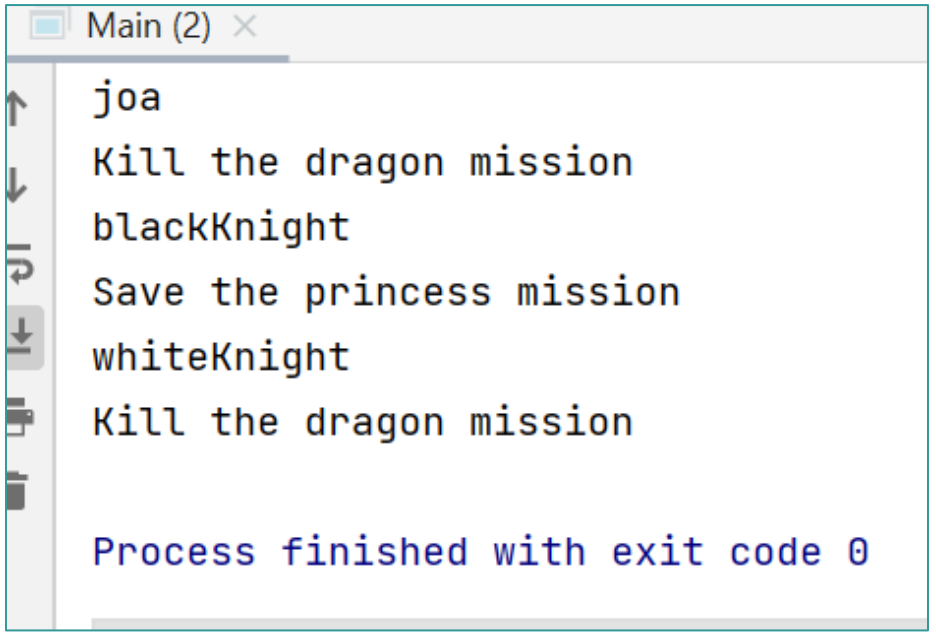
- Η run επειδή θα έχει πρόσβαση στον πίνακα, ο οποίος δεν είναι synchronized από την ίδια την Java, τον κάνουμε εμείς synchronized κάνοντάς τον monitored με το **synchronized** keyword ενώ στο synchronized block του κώδικα έχουμε το **critical section**

Main


```java
package gr.aueb.cf.tmp.sync;

public class Main {

    private final static Runnable joa = new Knight("joa");
    private final static Runnable whiteKnight = new Knight("whiteKnight");
    private final static Runnable blackKnight = new Knight("blackKnight");

    public static void main(String[] args) {
        (new Thread(joa)).start();
        (new Thread((whiteKnight))).start();
        (new Thread(blackKnight)).start();
    }
}
```

Non-sync vs Synchronized





- Στην 1η περίπτωση έχουμε κάνει comment out το synchronized, ενώ στην 2η περίπτωση είναι synchronized

Cloneable (1)

- **Πρόβλημα**: Αντιγραφή αντικειμένου (αντιγραφή όλων των πεδίων bit-by-bit)
- Για να μπορούμε να αντιγράφουμε αντικείμενα μιας κλάσης με την **clone()**, θα πρέπει η κλάση **να κάνει implement το interface Cloneable** (Marker interface) και να υλοποιεί την Object.clone() (η οποία ανήκει στην κλάση Object ως protected)
- Διαφορετικά αν πάμε να αντιγράψουμε μια κλάση που δεν κάνει implement το Cloneable το JVM δημιουργεί την εξαίρεση **CloneNotSupportedException**
- Για να αντιγράψουμε στη συνέχεια πρέπει να καλέσουμε την .clone()

Cloneable (2)

- Η **`.clone()`** ελέγχει αν μία κλάση υλοποιεί το **`Interface Cloneable`** ώστε να μην μπορεί να κληθεί τυχαία και από κλάσεις που δεν είναι Cloneable
- Το **`Interface Cloneable`** όπως και το Runnable και άλλα Interfaces της java που έχουν κατάληξη –able είναι ***mixin*** Interfaces δηλαδή προσθέτουν λειτουργικότητα στους τύπους των κλάσεων που τα υλοποιούν

Shallow vs Deep Copy

- Πρέπει να προσέξουμε το εξής όταν θέλουμε να κάνουμε αντιγραφή.
- Θέλουμε καταρχάς ανεξάρτητα αντικείμενα, δηλαδή το αντίγραφο θέλουμε να είναι ένα ανεξάρτητο αντίγραφο του αρχικού αντικειμένου
- Αυτό επιτυγχάνεται με την Object.clone() μόνο όταν τα πεδία της κλάσης είναι πρωταρχικοί τύποι δεδομένων ή immutable όπως τα Strings, διαφορετικά αν τα πεδία μιας κλάσης είναι αναφορές σε mutable αντικείμενα τότε **η Object.clone() δημιουργεί shallow copies** δηλαδή αντιγράφονται οι αναφορές των αντικειμένων και όχι τα περιεχόμενα, οπότε και πρέπει να γίνει override με διαφορετικό τρόπο, ώστε να αντιγράφει και τα περιεχόμενα (deep copy)

Κλάση City


```java
package gr.aueb.cf.clone;

public class City implements Cloneable {
    private String description;

    public City(String name) {
        this.description = name;
    }

    public String getDescription() { return description; }

    public void setDescription(String description) { this.description = description; }

    @Override
    protected City clone() throws CloneNotSupportedException {
        return (City) super.clone();
    }

    @Override
    public String toString() {
        return "City{" +
                "name='" + description + '\'' +
                '}';
    }
}
```

- Το πεδίο name είναι String που είναι immutable οπότε δουλεύει η **super.clone()** δηλαδή η Object.clone() που δημιουργεί ένα αντικείμενο City και κάνει αντιγραφή του πεδίου description δηλαδή του δείκτη στο description

Main


```java
package gr.aueb.cf.clone;

public class Main {

    public static void main(String[] args) throws CloneNotSupportedException {
        City athens = new City("Athens");

        City clonedAthens = athens.clone();

        clonedAthens.setDescription("Athens2");

        System.out.println(athens);
        System.out.println(clonedAthens);

    }
}
```

```
Main (3)
"C:\Program Files\Amazon Corretto\
City{name='Athens'}
City{name='Athens2'}

Process finished with exit code 0
```

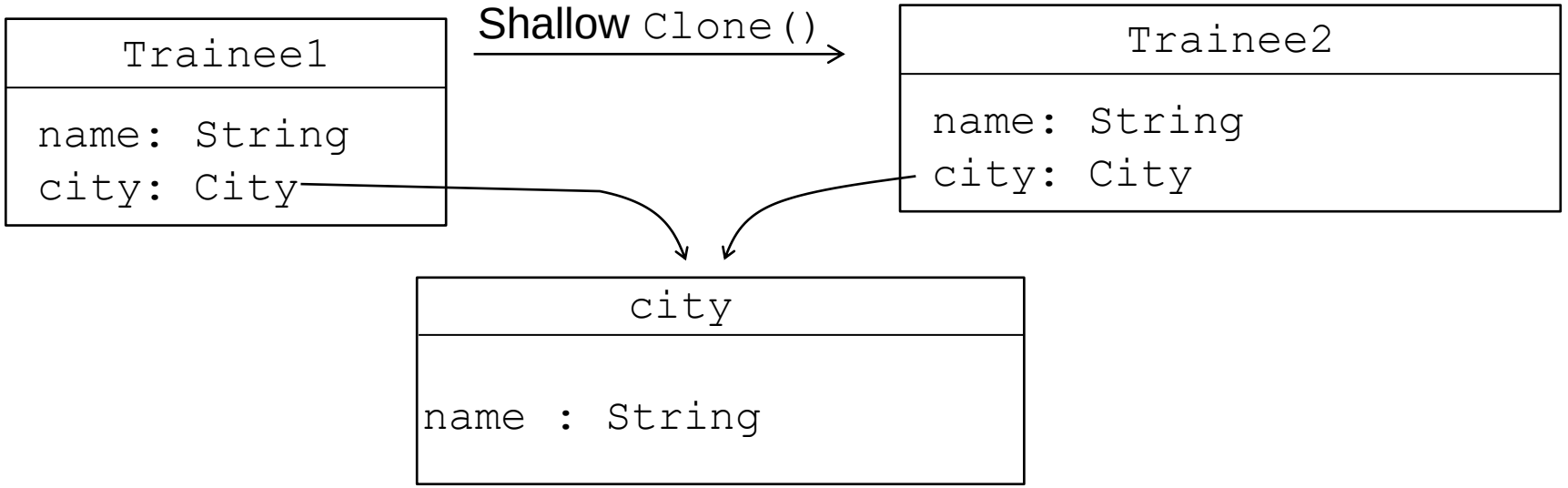
- Λειτουργεί σωστά λόγω immutability του description

Shallow Copy


```java
package gr.aueb.elearn.traineecloneshallow;

public class Trainee implements Cloneable {
    private String name;
    private City city;

    Trainee() {}

    Trainee(String name, City city) {
        this.name = name;
        this.city = city;
    }

    @Override
    protected Trainee clone() throws CloneNotSupportedException {
        return (Trainee) super.clone();
    }
```

- Η κλάση Trainee περιέχει ένα String και ένα αντικείμενο City. Εδώ η clone που απλά καλεί την *super.clone()* δεν λειτουργεί σωστά αφού δημιουργεί shallow copy γιατί αντιγράφεται το city ως αναφορά (δείκτης)

Shallow Copy



- Στο shallow copy όταν κάνουμε αντιγραφή τα πεδία city και των δύο αντικειμένων (αρχικού και αντιγράφου) δείχνουν στο ίδιο πραγματικό αντικείμενο city.
- Επομένως οποιαδήποτε αλλαγή στο description θα αντανακλάται και στο `Trainee1` και στο `Trainee2` και επομένως τα δύο αντικείμενα δεν είναι ανεξάρτητα

Deep Copy


```
28      @Override
29      protected Trainee clone() throws CloneNotSupportedException {
30          Trainee trainee = (Trainee) super.clone();
31          City city = new City(this.getCity().getDescription());
32          trainee.setCity(city);
33          return trainee;
34      }
```

- Αλλάζουμε την υλοποίηση της clone() Εδώ αφού γίνει η αντιγραφή με Object.clone() στη συνέχεια δημιουργούμε νέο city με new City και θέτουμε ως παράμετρο το description του this

Deep cloning Demo


```java
package gr.aueb.cf.clone;

public class Main {

    public static void main(String[] args) throws CloneNotSupportedException {
        Trainee alice = new Trainee("Alice", new City("Athens"));
        Trainee clonedTrainee = alice.clone();
        clonedTrainee.setCity(new City("Athens2"));

        System.out.println(alice);
        System.out.println(clonedTrainee);
    }
}
```

```
Main (3)
"C:\Program Files\Amazon Corretto\jdk11.0.10_9\b
Trainee{name='Alice', city=City{name='Athens'}}
Trainee{name='Alice', city=City{name='Athens2'}}

Process finished with exit code 0
```

- Παρατηρούμε ότι ότι τα δύο αντικείμενα είναι ανεξάρτητα. Αλλαγές στον *clonedTrainee* δεν επηρεάζουν την alice

Copy Constructors

- Όταν θέλουμε να κάνουμε αντιγραφή **θα ήταν προτιμότερο να χρησιμοποιούμε copy constructors** γιατί είναι απλούστεροι από την clone()
- Η clone() απαιτεί να κάνουμε implement το interface Cloneable **κάνοντας τον κώδικά μας invasive**, δηλαδή μας αναγκάζει να υλοποιήσουμε ένα Java-based interface κάνοντας τον κώδικά μας λιγότερο απλό, και λιγότερο επαναχρησιμοποιήσιμο δημιουργώντας εξαρτήσεις

City – Copy Constructor


```java
package gr.aueb.cf.tmp.clonedemo;

import java.util.Objects;

public class City{
    private String description;

    public City(String description) {
        this.description = description;
    }

    // Copy Constructor
    public City(City city) {
        this.description = city.description;
    }

    public String getDescription() { return description; }

    public void setDescription(String description) { this.description = description; }

    @Override
    public String toString() {...}
```

- Ο copy constructor είναι constructor που παίρνει ως παράμετρο ένα αντικείμενο ίδιου τύπου

Trainee – Copy Constructor


```java
package gr.aueb.cf.tmp.clonedemo;

import java.util.Objects;

public class Trainee {
    private String name;
    private City city;

    public Trainee(String name, City city) {
        this.name = name;
        this.city = new City(city);
    }

    // Copy Constructor
    public Trainee(Trainee trainee) {
        this.name = trainee.name;
        this.city = new City(trainee.getCity());
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public City getCity() { return city; }
    public void setCity(City city) { this.city = city; }
    @Override
    public String toString() {...}
```

- Εδώ ο copy constructor αρχικοποιεί με new το πεδίο city γιατί είναι αναφορική μεταβλητή (αντικείμενο mutable)

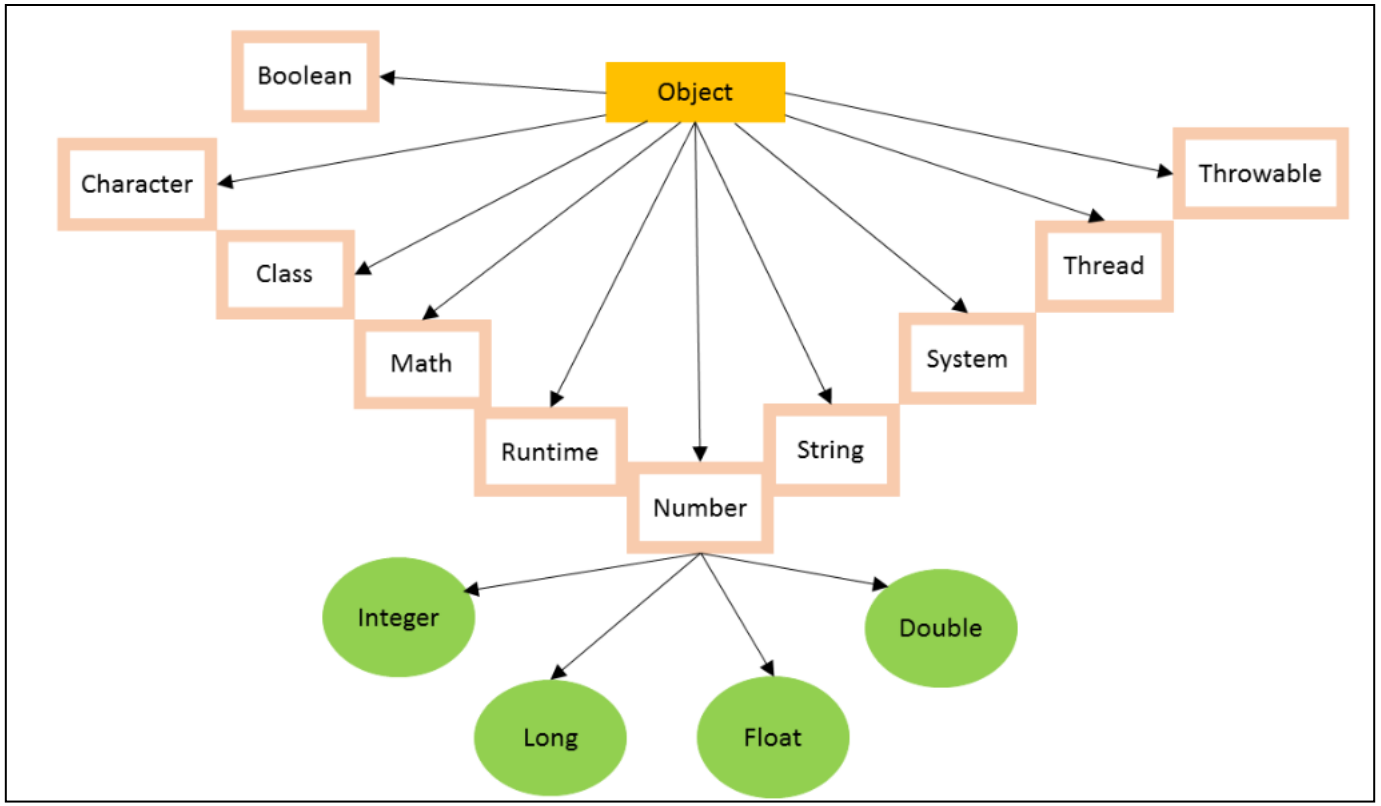
Copy Constructor Demo


```java
package gr.aueb.cf.tmp.clonedemo;

public class Main {

    public static void main(String[] args) {

        Trainee alice = new Trainee("Alice", new City("Athens"));
        Trainee bob = new Trainee(alice);

        bob.getCity().setDescription("New York");

        System.out.println(alice);
        System.out.println(bob);
    }
}
```

- Παρατηρήστε κι εδώ ότι τα δύο αντικείμενα είναι ανεξάρτητα. Αλλαγές στον trainee bob δεν επηρεάζουν την trainee alice

Κλάση Object

- Όπως έχουμε πει, η κλάση Object είναι η ρίζα της ιεραρχίας κληρονομικότητας της Java, από όπου κληρονομούν όλες οι κλάσεις της Java



Object Utility Methods

- ***toString()*** – επιστρέφει το hashCode σε Hex μορφή
- ***equals()*** – επιστρέφει true αν δύο αντικείμενα δείχνουν στην ίδια διεύθυνση μνήμης
- ***hashCode()*** – επιστρέφει τη διεύθυνση μνήμης ενός αντικειμένου
- Οι παραπάνω μέθοδοι της κλάσης Object όταν δεν υπερκαλύπτονται έχουν το παραπάνω default implementation με βάση τη διεύθυνση μνήμης των αντικειμένων που τις καλούν. **Γιαυτό θα πρέπει να τις κάνουμε override**
- Την toString() την έχουμε δει. Στη συνέχεια θα δούμε την equals() και την hashCode()

equals (1)

- Αν θέλουμε να ελέγξουμε την ισότητα δύο αντικειμένων Trainee με την equals τότε θα ελεγχθούν τα πεδία (fields) των αντικειμένων δηλαδή το name και το city
- Το city όμως όπως είδαμε είναι δείκτης και ο έλεγχος με την equals ελέγχει την τιμή του δείκτη (διεύθυνση μνήμης) και όχι το περιεχόμενο
- Αν δηλαδή κάνουμε deep copy και έχουμε δύο διαφορετικά αντικείμενα με διαφορετικούς δείκτες στο city **με ίδιο όμως περιεχόμενο**, τότε η equals θα επιστρέψει false, **ενώ θα έπρεπε να επιστρέψει true**

equals (2)


```
38      @Override
39      public boolean equals(Object o) {
40          if (this == o) return true;
41          if (o == null || getClass() != o.getClass()) return false;
42          Trainee trainee = (Trainee) o;
43          return Objects.equals(getName(), trainee.getName()) && Objects.equals(getCity(), trainee.getCity());
44      }
```

- Για να δουλέψει λοιπόν σωστά η equals σε αντικείμενα που περιέχουν άλλα αντικείμενα (δηλ. αναφορές σε άλλα αντικείμενα) θα πρέπει να υπερκαλυφθεί όπως εδώ, δηλαδή να ελέγξουμε το περιεχόμενο των πεδίων των αντικειμένων
- Η μέθοδος Objects.equals(Object a, Object b) ελέγχει με τη χρήση της equals τα a, b αν είναι ίσα. Αν τα a, b είναι String τότε καλείται η equals της κλάσης String που είναι ήδη overridden. Διαφορετικά, αν πρόκειται για objects -όπως εδώ- θα πρέπει να υπερκαλυφθεί η equals και στην κλάση City, διαφορετικά θα χρησιμοποιηθεί η default equals που ελέγχει αναφορές αντικειμένων. Θα πρέπει λοιπόν να υπερκαλυφθεί η equals() στην City απλά καλώντας την String.equals()

City equals


```
25      @Override
26      public boolean equals(Object o) {
27          if (this == o) return true;
28          if (o == null || getClass() != o.getClass()) return false;
29          City city = (City) o;
30          return getDescription().equals(city.getDescription());
31      }
```

- Υπερκαλύπτουμε την equals στην κλάση City ώστε να ελέγχει τα descriptions

hashCode()

- Μία **Hash Function** είναι μία συνάρτηση που μπορεί με βάση μία τιμή (**key**) να δώσει ένα μοναδικό αποτέλεσμα (Hash code, Hash value ή απλά **Hash**)
- Όπως οι μαθηματικές συναρτήσεις, για κάθε διαφορετική τιμή, το αποτέλεσμα πρέπει να είναι διαφορετικό και για ίδιες τιμές πρέπει το αποτέλεσμα να είναι ίδιο

equals() και hashCode()

- Στο παράδειγμα με τα αντικείμενα Trainee ακολουθούμε τον γενικό κανόνα πως **όταν υπερφορτώνουμε την equals(), τότε θα πρέπει να υπερφορτώνουμε και την hashCode()** ώστε δύο αντικείμενα που είναι 'ίσα' με βάση την equals() να έχουν και ίδιο hash-code ή αν δεν είναι 'ίσα' να έχουν διαφορετικό hash-code

City


```
32      @Override
33      public boolean equals(Object o) {
34          if (this == o) return true;
35          if (o == null || getClass() != o.getClass()) return false;
36          City city = (City) o;
37          return getDescription().equals(city.getDescription());
38      }
39
40      @Override
41      public int hashCode() {
42          return Objects.hash(getDescription());
43      }
44  }
```

Objects.hash()


```
46      @Override
47      public int hashCode() {
48          return Objects.hash(getName(), getCity());
49      }
```

- Η utility κλάση της Java, Objects, παρέχει μία static μέθοδο Objects.hash() που λαμβάνει μία μεταβλητή λίστα αντικειμένων τύπου Object (άρα οποιοδήποτε αντικείμενο μιας και όλα τα αντικείμενα κληρονομούν από την Object) και επιστρέφει το hash code.
- Η μέθοδος hash της κλάσης Objects καλεί στην πραγματικότητα την μέθοδο Arrays.hashCode() που περιλαμβάνει έναν πίνακα στον οποίο αποθηκεύονται οι παράμετροι της Objects.hash() καθώς και κάνει boxing/unboxing παραμέτρων πρωταρχικού τύπου
- Για τους παραπάνω λόγους **είναι αργή και προτείνεται μόνο για εφαρμογές που η απόδοση δεν είναι κρίσιμος παράγοντας**

Trainee


```
40      @Override
41      public boolean equals(Object o) {
42          if (this == o) return true;
43          if (o == null || getClass() != o.getClass()) return false;
44          Trainee trainee = (Trainee) o;
45          return getName().equals(trainee.getName()) &&
46                  Objects.equals(getCity(), trainee.getCity());
47      }
48
49      @Override
50      public int hashCode() {
51          return Objects.hash(getName(), getCity());
52      }
```

Overridden hashCode()


```
39⊖    @Override
40     public int hashCode() {
41         final int prime = 31;
42         int result = 1;
43         result = prime * result + ((city == null) ? 0 : city.hashCode());
44         result = prime * result + ((name == null) ? 0 : name.hashCode());
45         return result;
46     }
```

- Σε αυτή την έκδοση της hashCode() που έχουμε υπερφορτώσει εφαρμόζεται ένας αλγόριθμος συνάρτησης hash με βάση ένα πρώτο αριθμό το 31 και με βάση τα hash codes των πεδίων της κλάσης Trainee
- Η απόδοση εδώ είναι πολύ γρήγορη
- Θα πρέπει να υπερκαλύψουμε και την hashCode της City με τον ίδιο τρόπο με *description.hashCode()*
- Η κλάση String έχει υπερκαλύψει την hashCode() ως $s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + s[2]*31^{(n-3)} + \ldots + s[n-1]*31^{(0)}$

Serializable (1)

- Serialization σημαίνει την **μετατροπή ενός αντικειμένου σε byte-stream**. Δηλαδή η εγγραφή ενός αντικειμένου σε μία ροή εξόδου **ObjectOutputStream** ονομάζεται serialization.
- Το byte-stream μπορεί να επανακτηθεί από μία ροή εισόδου **ObjectInputStream** σε ένα αντίγραφο του αντικειμένου
- Για να μπορεί να γίνει αυτό σε ένα αντικείμενο θα πρέπει η κλάση του να υλοποιεί το **interface java.io.Serializable**

Serializable (2)

- **Serialized-form** είναι τα **πεδία που γίνονται export** όταν ένα αντικείμενο μιας κλάσης γίνεται serialize.
- Με το πέρασμα του χρόνου μπορεί να αλλάξει η υλοποίηση του serialized-from και να προστεθούν κι άλλα πεδία
- Κάτι τέτοιο όμως θα ήταν ασύμβατο με το παλιό serialized-form της κλάσης

Serializable (3)

- Για να μπορούμε να ξεχωρίζουμε τις εκδόσεις των serialized forms χρησιμοποιούμε ένα μοναδικό κωδικό serial Unique Identifier (UID) ορίζοντας ένα πεδίο serialVersionUID τύπου long **private static final long serialVersionUID;** που μπορούμε να του δώσουμε μία τιμή (για παράδειγμα 1 που να αυξάνει όταν αλλάζει η έκδοση)

Παράδειγμα Serialization


```java
package gr.aueb.elearn.chap12;

import java.io.Serializable;
import java.util.Objects;

public class Trainee implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private City city;
    private transient int hashcode;
```

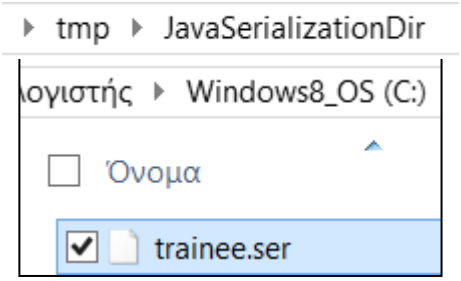
- Τα `transient` πεδία δεν γίνονται serialize

```java
package gr.aueb.elearn.serialize;

import java.io.Serializable;

public class City implements Serializable {
    private static final long serialVersionUID = 1L;
    private String description;
```

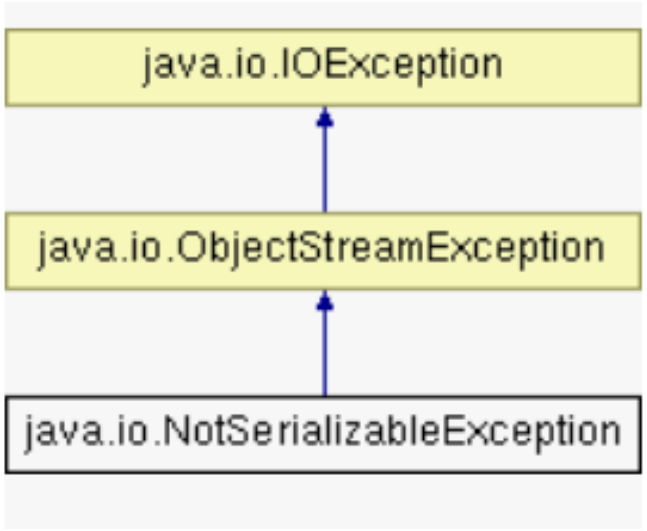
Serialization Demo


```
1  package gr.aueb.elearn.serialize;
2
3  import java.io.*;
4
5  public class TraineeWriteDemo {
6
7      public static void main(String[] args) {
8
9          try (ObjectOutputStream oos = new ObjectOutputStream(
10                 new FileOutputStream("C:/tmp/JavaSerializationDir/trainee.ser"))) {
11             Trainee traineeAlice = new Trainee("Alice", new City("Athens"));
12             oos.writeObject(traineeAlice);
13             System.out.println(traineeAlice.getName() + " successfully saved");
14         } catch (FileNotFoundException e) {
15             e.printStackTrace();
16         } catch (NotSerializableException e1) {
17             System.out.println("Not Serializable");
18         } catch (IOException e2) {
19             System.out.println(e2.getMessage());
20         }
21     }
22 }
```

- Δημιουργούμε το αντικείμενο traineeAlice
- Δημιουργούμε ένα **ObjectOutputStrem**
- Κάνουμε **writeObject(traineeAlice)**



NotSerializableException



- Η μέθοδος **ObjectOutputStream.writeObject()**, που κάνει serialize ένα αντικείμενο που λαμβάνει ως παράμετρο απαιτεί η παράμετρος (δηλαδή το αντικείμενο εισόδου) να είναι serializable αλλιώς δημιουργείται μία εξαίρεση **NotSerializableException**

Deserialization Demo


```
1  package gr.aueb.elearn.serialize;
2
3  import java.io.*;
4
5  public class TraineeReadDemo {
6
7      public static void main(String[] args) {
8          try (ObjectInputStream ois = new ObjectInputStream(
9                  new FileInputStream("C:/tmp/JavaSerializationDir/trainee.ser"))) {
10             Trainee traineeAlice;
11             traineeAlice = (Trainee) ois.readObject();
12             System.out.println(traineeAlice.getName() + " successfully read");
13         } catch (FileNotFoundException e) {
14             e.printStackTrace();
15         } catch (NotSerializableException | ClassNotFoundException e1) {
16             System.out.println("Not Serializable");
17         } catch (IOException e2) {
18             System.out.println(e2.getMessage());
19         }
20     }
21 }
```

- Διαβάζουμε με *ObjectInputStream* που λαμβάνει ως παράμετρο *FileInputStream* και μετά διαβάζουμε με *.readObject*

Εργασία

- Στις κλάσεις της εργασίας του προηγούμενου κεφαλαίου προσθέστε τις equals και hashCode σε όλες τις κλάσεις της εφαρμογής, όπως Line, Circle, Rectangle
- Προσθέστε copy constructors
- Κάντε τις Serializable
- Κάντε τις Cloneable