

Φροντιστήριο JAVA

Abstract classes, Interfaces

Τι θα συζητήσουμε σήμερα

- Αφαιρέσεις στη Java
 - Abstract μέθοδοι και abstract κλάσεις
 - Interfaces (=διασυνδέσεις, διεπαφές)
 - Παραδείγματα κώδικα

Αφηρημένες μέθοδοι

- Στη Java μπορούμε να *δηλώσουμε* ένα αντικείμενο χωρίς να το *ορίσουμε* (κατασκευάσουμε):

Person p;

- Ανάλογα, μπορούμε να *δηλώσουμε* μια μέθοδο, χωρίς να την *ορίσουμε*:

public abstract **void** draw(int size);

- Υπάρχει η υπογραφή και ο επιστρεφόμενος τύπος αλλά λείπει το σώμα (η υλοποίηση) της μεθόδου
- Μια μέθοδος που έχει δηλωθεί αλλά δεν έχει οριστεί ονομάζεται **abstract method**

Αφηρημένες κλάσεις 1/2

- Οποιαδήποτε κλάση περιλαμβάνει μια αφηρημένη μέθοδο είναι και αυτή αφηρημένη (abstract class)
- Πρέπει να ορίζουμε την κλάση με το keyword **abstract**:

```
abstract class MyClass {...}
```
- Μια αφηρημένη κλάση είναι ημιτελής
 - Της λείπουν κάποια από τα σώματα των μεθόδων που περιέχει
- **Δε μπορούμε να δημιουργήσουμε στιγμιότυπα (αντικείμενα) από αφηρημένες κλάσεις**

Αφηρημένες κλάσεις 2/2

- Μπορούμε να κάνουμε **extend** (δηλ. να ορίσουμε υποκλάσεις) σε μια abstract class
 - Αν η υποκλάση ορίζει το σώμα όλων των αφηρημένων μεθόδων που κληρονόμησε, είναι «πλήρης» (concrete) και μπορεί να παράγει αντικείμενα
 - Αν ΌΧΙ, τότε και η υποκλάση θα πρέπει να δηλωθεί ως αφηρημένη (abstract)
- Μπορούμε να ορίσουμε μια κλάση ως **abstract** ακόμα κι αν *δεν περιέχει κάποια abstract μέθοδο*
 - Έτσι εμποδίζουμε τη δημιουργία αντικειμένων από την κλάση αυτή

Γιατί χρειαζόμαστε abstract classes?

- Έστω ότι θέλουμε να δημιουργήσουμε μια κλάση **Shape**, με υποκλάσεις τις **Oval**, **Rectangle**, **Triangle**, **Hexagon**, **Star** κτλ.
- Και δε θέλουμε να επιτρέπεται η δημιουργία αντικειμένων τύπου Shape
 - Έχει νόημα να υπάρχουν μόνο αντικείμενα συγκεκριμένων σχημάτων
 - Αν η **Shape** είναι abstract, δε μπορεί να δημιουργηθεί ένα αντικείμενο **Shape**
 - Μπορεί όμως να δημιουργηθεί ένα αντικείμενο **Oval**, ή **Rectangle**, κτλ.
- Οι abstract κλάσεις είναι καλές για να ορίζουν μια γενική κατηγορία που περιέχει συγκεκριμένες υποκλάσεις.

Παράδειγμα abstract class

- ```
public abstract class Animal {
 abstract int eat();
 abstract void breathe();
}
```

- Δε μπορούμε να δημιουργήσουμε αντικείμενα της κλάσης Animal
- Οποιαδήποτε μη αφηρημένη υποκλάση της Animal πρέπει να παρέχει υλοποιήσεις των μεθόδων `eat()` και `breathe()`

# Γιατί υπάρχουν abstract methods?

- Έστω η μη αφηρημένη κλάση **Shape**
  - Η **Shape** δεν πρέπει να έχει τη μέθοδο **draw()**
  - Κάθε υποκλάση της **Shape** πρέπει να έχει μέθοδο **draw()** method
- Τώρα ας υποθέσουμε ότι έχουμε μια μεταβλητή **figure** που παίρνει σαν τιμή αντικείμενα τύπου Shape και ότι περιέχει κάποιο αντικείμενο υποκλάσης της Shape (π.χ. της κλάσης **Star**), δηλ.
  - `Shape figure = new Star();`
  - Αν γράψουμε `figure.draw()`, θα προκληθεί syntax error επειδή ο compiler δε μπορεί να γνωρίζει τι είδους τιμή θα δοθεί στην μεταβλητή **figure**
  - Μια κλάση γνωρίζει την superclass στην οποία ανήκει αλλά δεν γνωρίζει τις υποκλάσεις της



# Γιατί υπάρχουν abstract methods?

- **Λύση:** Ορίζουμε στην **Shape** μια *abstract* μέθοδο **draw()**
  - Τώρα η **Shape** είναι abstract, και δε μπορεί να παράγει αντικείμενα
  - Η μεταβλητή **figure** δε μπορεί να περιέχει αντικείμενο της **Shape** γιατί είναι αδύνατο να δημιουργηθεί τέτοιο αντικείμενο
  - Οποιοδήποτε αντικείμενο (όπως ένα αντικείμενο της κλάσης **Star**) που ανήκει σε υποκλάση της **Shape** θα διαθέτει μια μέθοδο **draw()**
  - Ο compiler της Java μπορεί να είναι σίγουρος ότι η κλήση **figure.draw()** θα είναι σε κάθε περίπτωση νόμιμη και δεν παράγει syntax error

# Το πρόβλημα που περιγράψαμε

```
class Shape { ... }
```

```
class Star extends Shape {
 void draw() { ... }
 ...
}
```

```
class Circle extends Shape {
 void draw() { ... }
 ...
}
```

```
Shape someshape = new Star();
```

Νόμιμο αφού ένα αντικείμενο Star είναι και αντικείμενο Shape

```
someShape.draw();
```

- Προκαλείται syntax error, γιατί ένα αντικείμενο Shape μπορεί να μην έχει μέθοδο draw()
- Remember: ***A class knows its superclass, but not its subclasses***

# Μια λύση

```
abstract class Shape {
 abstract void draw();
}
class Star extends Shape {
 void draw() { ... }
 ...
}
class Circle extends Shape {
 void draw() { ... }
 ...
}
```

```
Shape someShape = new Star();
```

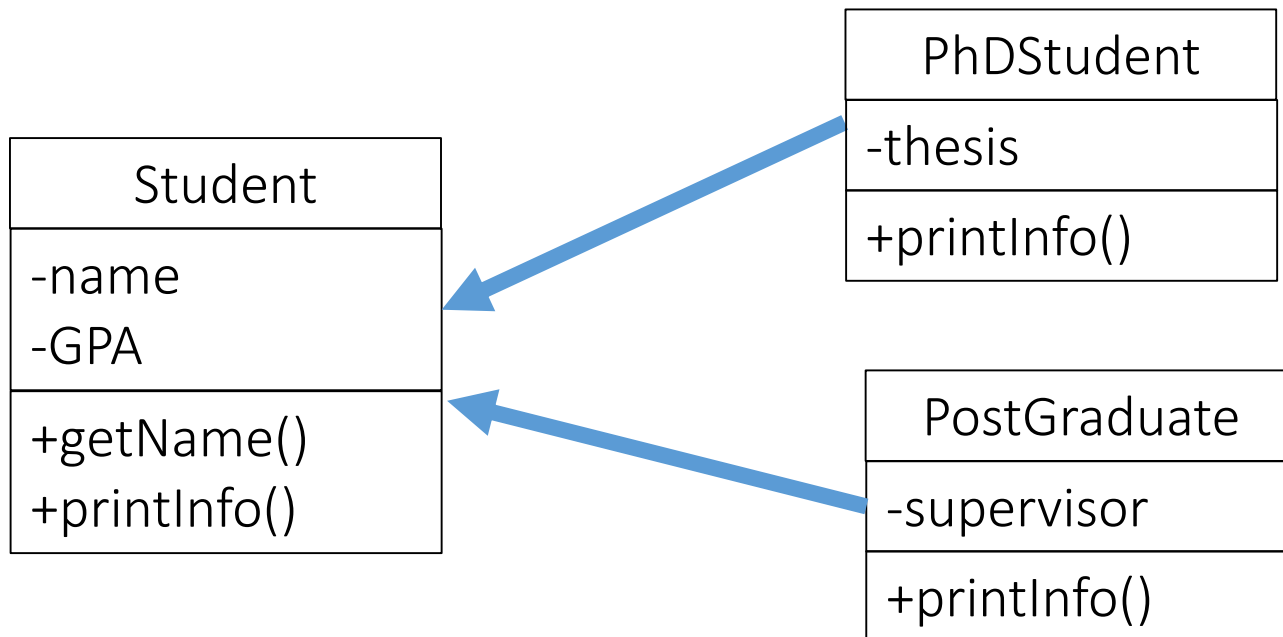
- Νόμιμο επειδή ένα αντικείμενο Star είναι και αντικείμενο Shape
- Όμως τώρα το `Shape someShape = new Shape();` δεν επιτρέπεται

```
someShape.draw();
```

- Νόμιμο, γιατί κάθε στιγμιότυπο που μπορεί να παραχθεί θα έχει υποχρεωτικά μια μέθοδο `draw()`

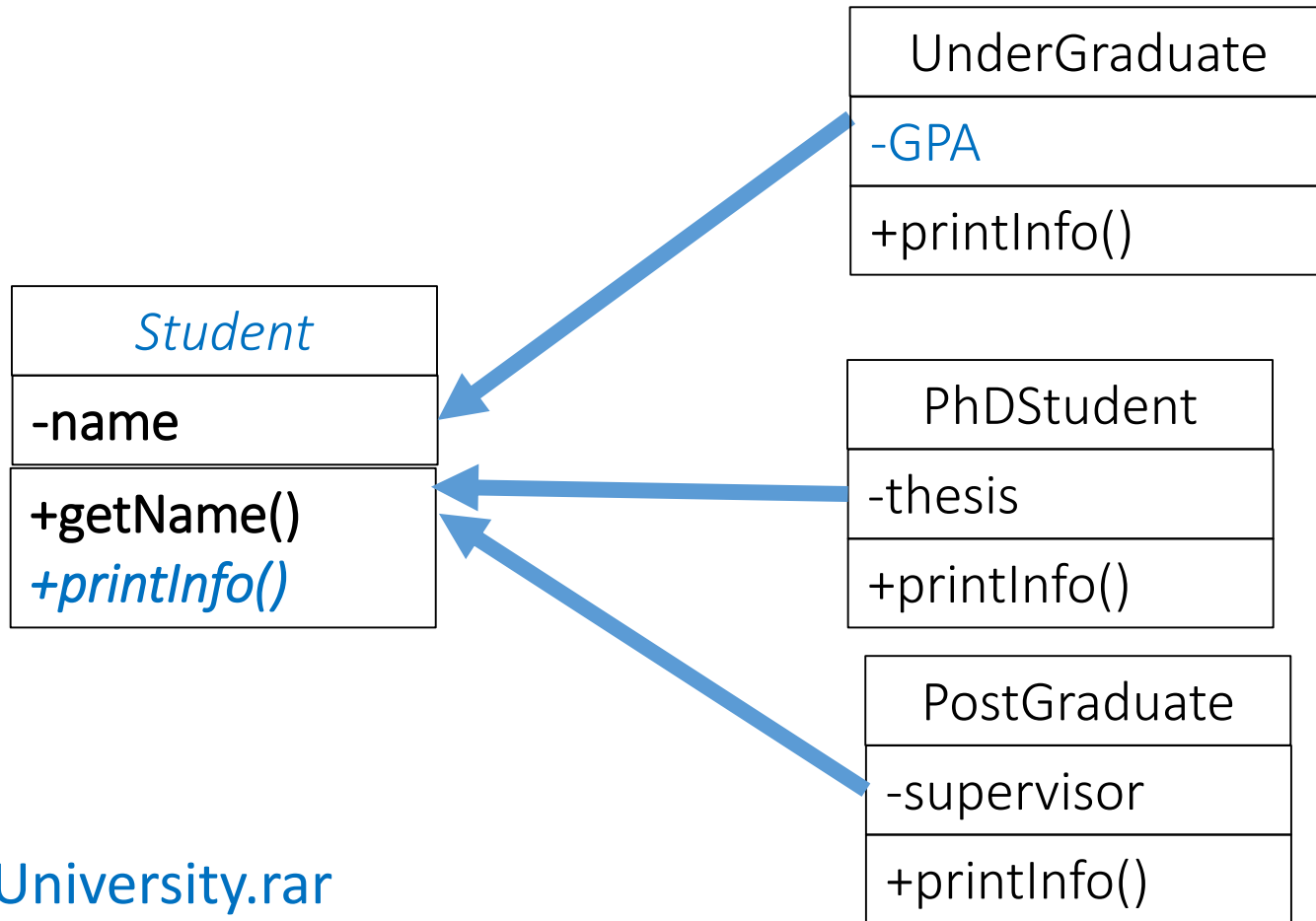
# Ας δούμε ένα άλλο παράδειγμα

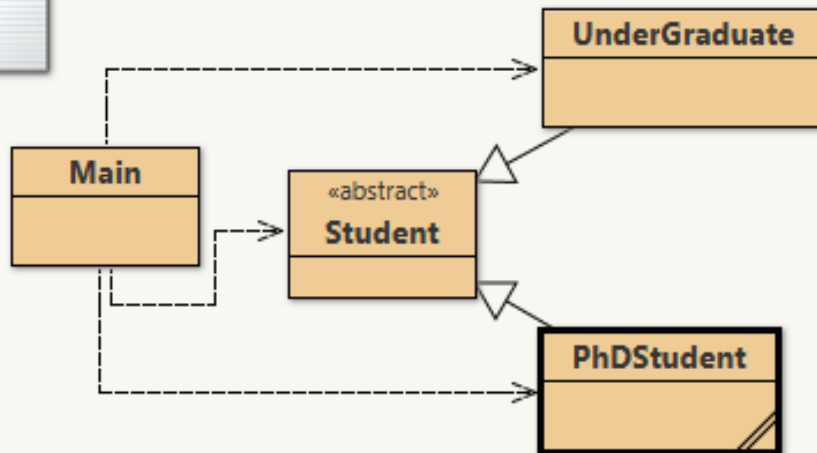
- Έστω ότι θέλουμε να μοντελοποιήσουμε σε ένα σύστημα τις οντότητες προπτυχιακός φοιτητής (Student), μεταπτυχιακός (PostGraduate) και διδακτορικός φοιτητής (PhDStudent)



- Αν πρέπει να αλλάξουμε τη σχεδίαση ώστε μόνο ο προπτυχιακός φοιτητής να έχει GPA (grade point average)?

# Χρήση abstract κλάσης





```

public abstract class Student {

 private String name;

 public Student(String name) {
 this.name = name; }

 public String getName() {
 return name;
 }

 public abstract void printInfo();

```

```

import java.util.ArrayList;

public class Main {

 public static void main(String[] args) {

 ArrayList<Student> students = new ArrayList<Student>();

 students.add(new UnderGraduate("Giorgos", 7.78));
 students.add(new PhDStudent("Nikos", "Spacial Data Structures"));
 students.add(new UnderGraduate("Eleni", 8.2));

 for(Student student: students)
 student.printInfo();
 }

 }

```

```

public class UnderGraduate extends Student {

 private double GPA;

 public UnderGraduate(String name, double gPA) {
 super(name);
 GPA = gPA;
 }

 public void printInfo() {
 System.out.println("Undergraduate Student");
 System.out.println("Name: " + getName());
 System.out.println("GPA: " + GPA);
 System.out.println();
 }
}

```

```

public class PhDStudent extends Student {

 private String thesis;

 public PhDStudent(String name, String thesis) {
 super(name);
 this.thesis = thesis;
 }

 public void printInfo() {
 System.out.println("PhD Student");
 System.out.println("Name: " + getName());
 System.out.println("Thesis: " + thesis);
 System.out.println();
 }
}

```

```
import java.util.ArrayList;

public class Main {

 public static void main(String[] args) {

 ArrayList<Student> students = new ArrayList<Student>();

 students.add(new UnderGraduate("Giorgos", 7.78));
 students.add(new PhDStudent("Nikos", "Spacial Data Structures"));
 students.add(new UnderGraduate("Eleni", 8.2));

 for(Student student: students)
 student.printInfo();
 }
}
```

**Undergraduate Student**

**Name: Giorgos**

**GPA: 7.78**

**PhD Student**

**Name: Nikos**

**Thesis: Spacial Data Structures**

**Undergraduate Student**

**Name: Eleni**

**GPA: 8.2**



# Interfaces (Διασυνδέσεις)

- Ένα `interface` δηλώνει (προδιαγράφει) μεθόδους αλλά δεν παρέχει το σώμα τους (δεν τις υλοποιεί)

```
interface KeyListener {
 public void keyPressed(KeyEvent e);
 public void keyReleased(KeyEvent e);
 public void keyTyped(KeyEvent e);
}
```

- Όλες οι μέθοδοι είναι έμμεσα `public` και `abstract`
  - Μπορούμε και να το δηλώνουμε ρητά αλλά δεν υπάρχει λόγος!
- Δε μπορούμε να δημιουργήσουμε αντικείμενα από ένα `interface`
  - Το `interface` είναι σαν μια `πολύ abstract κλάση` — δεν υλοποιεί `καμία` από τις μεθόδους του
- Μπορεί να περιέχει και σταθερές (`final` μεταβλητές). Συντακτικά επιτρέπονται και μεταβλητές αλλά συνήθως δεν έχει νόημα να τις δηλώνουμε (μετατρέπονται σε `static final`)

<http://tutorials.jenkov.com/java/interfaces.html>

*The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. Class KeyEvent. An event which indicates that a keystroke occurred in a component.*

# Σχεδίαση interfaces

- Συνήθως χρησιμοποιούμε **έτοιμα interfaces** Java που μας παρέχει η Oracle
- Σε κάποιες περιπτώσεις μπορεί να χρειαστεί να σχεδιάσουμε δικά μας interfaces
  - Π.χ. όταν θέλουμε κλάσεις διαφορετικού είδους να έχουν κάποιες κοινές δυνατότητες (μεθόδους)
- Αν θέλουμε να μπορούμε να δημιουργούμε animated απεικονίσεις όλων των αντικειμένων μια κλάσης θα ορίζαμε ένα interface:

```
public interface Animatable {
 install(Panel p);
 display();
}
```

- Έτσι μπορούμε να γράψουμε κώδικα που εμφανίζει οποιοδήποτε αντικείμενο κλάσης **Animatable** σε ένα **Panel** της επιλογής μας, απλά καλώντας αυτές τις μεθόδους

*Panel is the simplest container class. A panel provides space in which an application can attach any other component, including other panels.*

# Υλοποίηση ενός interface 1/2

- Μπορούμε να κάνουμε **extend** μια class, αλλά να υλοποιήσουμε (να κάνουμε **implement**) ένα interface
- Μια κλάση μπορεί να κάνει extend **μόνο μία** άλλη κλάση (δηλ. να είναι υποκλάση μιας μόνο κλάσης), αλλά μπορεί να υλοποιεί **όσα interfaces θέλουμε**
- Παράδειγμα:

```
class MyListener
 implements KeyListener, ActionListener { ... }
```

## Υλοποίηση ενός interface 2/2

- Όταν λέμε ότι μια κλάση υλοποιεί (**implements**) ένα interface, πρακτικά δίνουμε την υπόσχεση ότι θα ορίσουμε (θα υλοποιήσουμε) μέσα στην κλάση όλες τις μεθόδους που δηλώνονται στο interface

- Π.χ.:

```
class MyKeyListener implements KeyListener {
 public void keyPressed(KeyEvent e) {...};
 public void keyReleased(KeyEvent e) {...};
 public void keyTyped(KeyEvent e) {...};
}
```

- Στη θέση των “...” πρέπει να μπει ο κώδικας που υλοποιεί τις μεθόδους
- Στη συνέχεια μπορούμε να δημιουργήσουμε ένα αντικείμενο **MyKeyListener**

*The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. Class KeyEvent. An event which indicates that a keystroke occurred in a component.*

# Μερική υλοποίηση ενός Interface

- Μπορούμε να ορίσουμε μόνο κάποιες από τις μεθόδους ενός interface αλλά όχι όλες:

```
abstract class MyKeyListener implements KeyListener {
 public void keyTyped(KeyEvent e) {...};
}
```

- Αν μια κλάση δεν παρέχει τις υλοποιήσεις όλων των μεθόδων που υποσχέθηκε είναι μια **αφηρημένη** κλάση
- Άρα πρέπει να τη δηλώσουμε ως **abstract**
- Μπορούμε να επεκτείνουμε (*extend*) ένα interface (και έτσι να του προσθέσουμε μεθόδους):
  - interface FunkyKeyListener extends KeyListener { ... }

# Γιατί χρησιμοποιούμε interfaces;

- Λόγος 1: Μια κλάση μπορεί να **κληρονομεί** μόνο 1 άλλη κλάση αλλά μπορεί να **υλοποιεί** πολλά interfaces
  - Έτσι η κλάση μπορεί να παίζει διάφορους ρόλους (να υλοποιεί πολλές συμπεριφορές)
  - Ειδικά όταν υλοποιούμε Applets, είναι συνήθης πρακτική μια κλάση να υλοποιεί πολλούς διαφορετικούς listeners
  - Παράδειγμα:

```
class MyApplet extends Applet
 implements ActionListener, KeyListener {
 ...
}
```

- Λόγος 2: Μπορούμε να γράψουμε μεθόδους που χρησιμοποιούνται σε περισσότερες από μία κλάσεις
  - Από τη Java 8 μπορούμε να ορίσουμε και υλοποίηση μεθόδων σε ένα interface (χρησιμοποιώντας το keyword **default**)

# Πώς χρησιμοποιούμε τα interfaces

```
interface RuleSet {
 boolean isLegal(Move m, Board b);
 void makeMove(Move m);
}
```

Κάθε κλάση που υλοποιεί το Interface `RuleSet` πρέπει να διαθέτει αυτές τις δύο μεθόδους

```
class CheckersRules implements RuleSet { // Μία υλοποίηση
 public boolean isLegal(Move m, Board b) { ... }
 public void makeMove(Move m) { ... }
}
```

```
class ChessRules implements RuleSet { ... } // άλλη υλοποίηση
```

```
class LinesOfActionRules implements RuleSet { ... } // και άλλη μία
```

```
RuleSet rulesOfThisGame = new ChessRules();
```

Η εντολή ανάθεσης είναι νόμιμη γιατί ένα αντικείμενο `rulesOfThisGame` είναι αντικείμενο `RuleSet`

```
if (rulesOfThisGame.isLegal(m, b)) { makeMove(m); }
```

Νόμιμη εντολή γιατί ανεξάρτητα από το τι είδος αντικείμενο είναι το `rulesOfThisGame` θα διαθέτει τις μεθόδους `isLegal` και `makeMove`

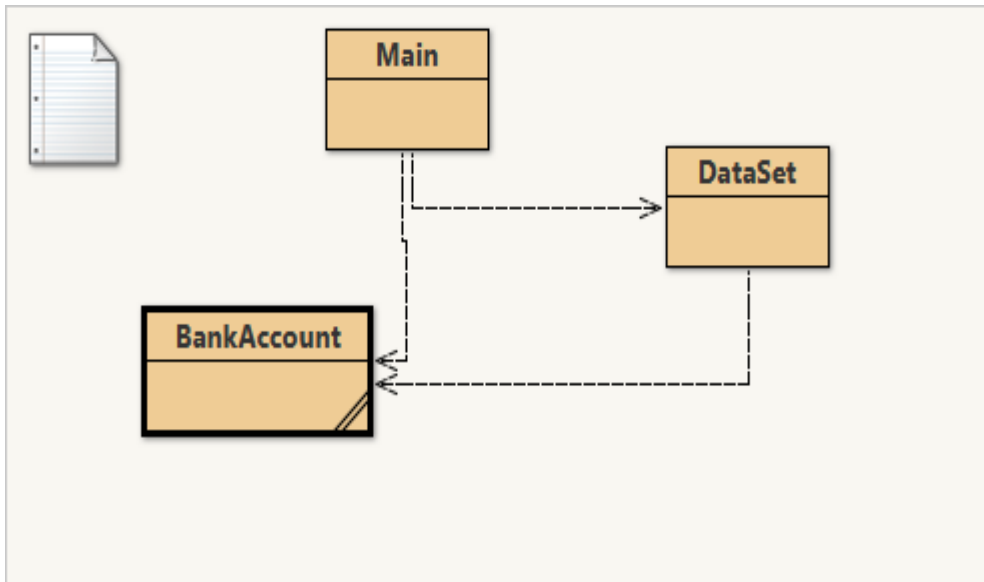
*Checkers, Chess, Lines of Action : τρία παιχνίδια*

# Ακόμα ένα παράδειγμα

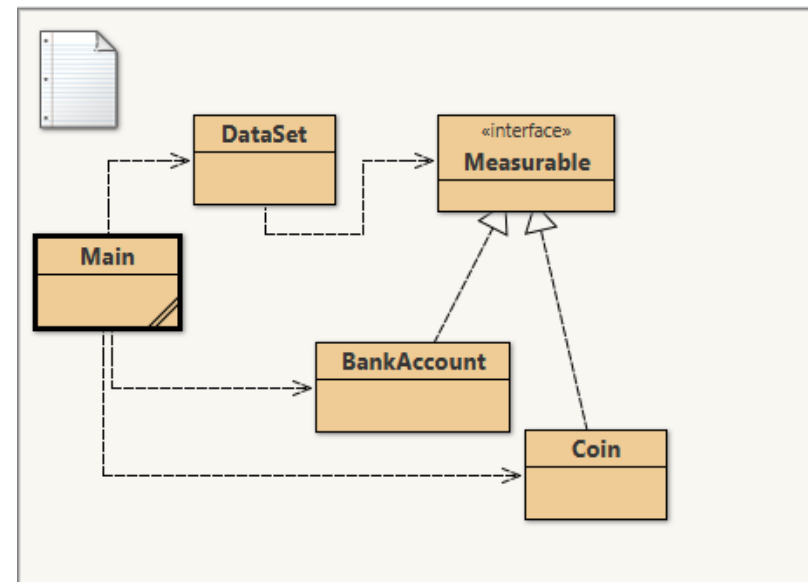
- Υλοποίηση και χρήση της κλάσης DataSet που λειτουργεί ως δομή υπολογισμού στατιστικών για τα στοιχεία που προσθέτω σε αυτή (count, sum, max, min)
  - Χωρίς χρήση αφαιρέσεων ([Dataset\\_noAbstractions.zip](#))
  - Με χρήση αφαιρέσεων ([Dataset\\_withAbstractions.zip](#))

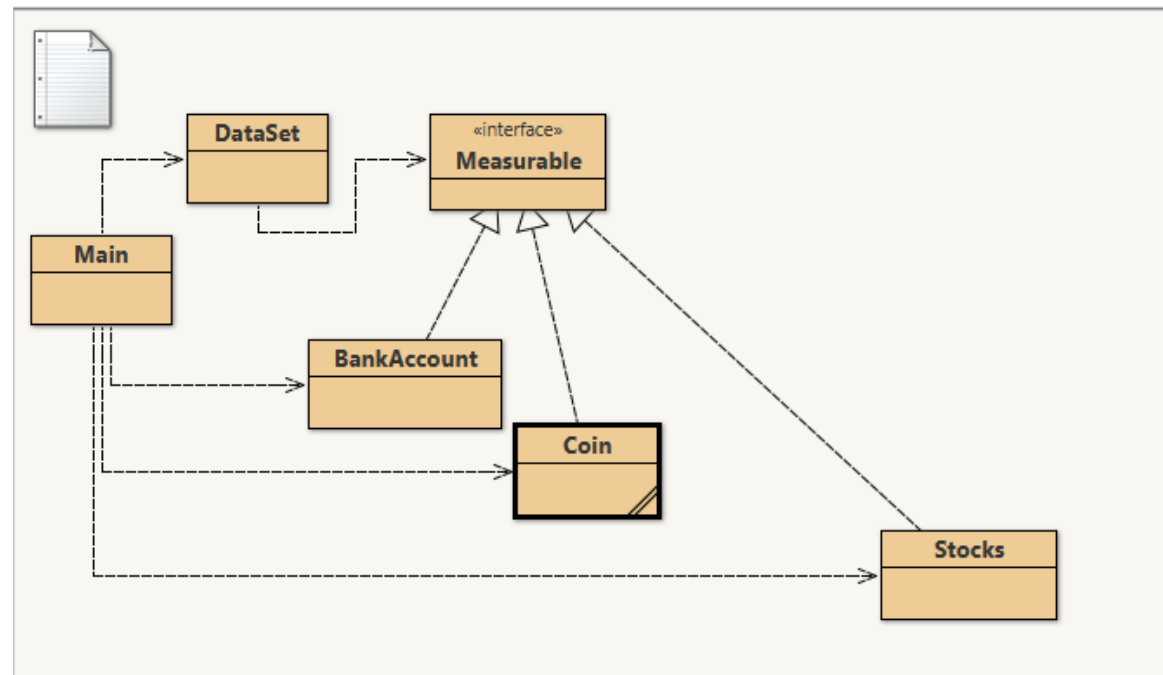
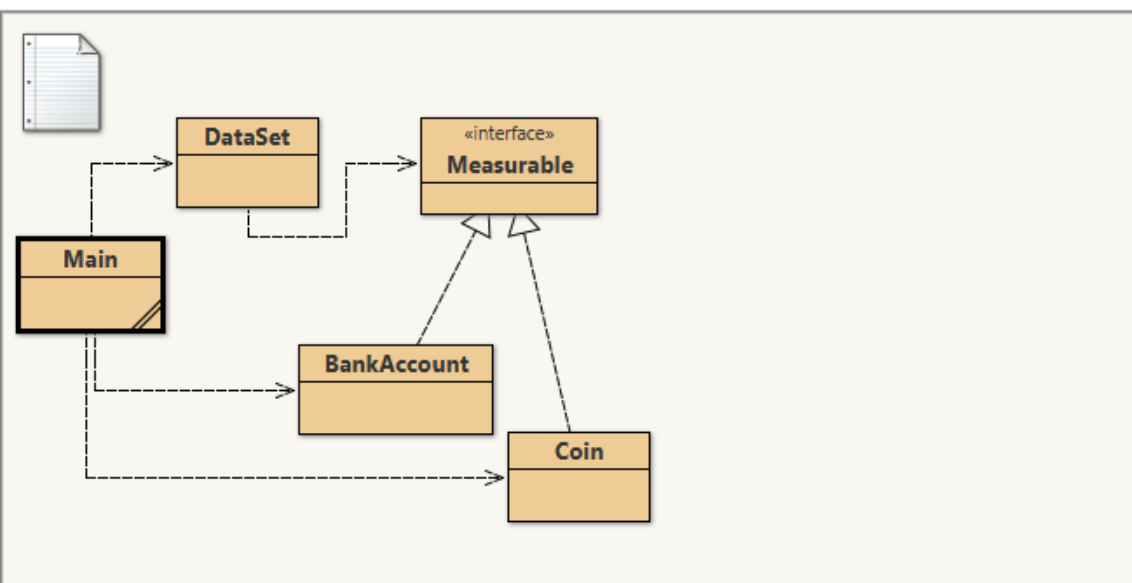


no\_abstraction



abstraction





# instanceof

- **instanceof** είναι ένα keyword (τελεστής) που απαντά στο αν μια μεταβλητή είναι μέλος μια κλάσης ή ενός Interface
  - Επιστρέφει true / false
- Παράδειγμα, αν δηλώσουμε ότι

```
class Dog extends Animal implements Pet {...}
Animal fido = new Dog();
```

Τότε τα παρακάτω είναι όλα true:

```
fido instanceof Dog
```

```
fido instanceof Animal
```

```
fido instanceof Pet
```

- Το **instanceof** σπάνια χρησιμοποιείται
  - Κάθε φορά που χρειάζεται να χρησιμοποιήσουμε το **instanceof**, θα πρέπει να σκεφτούμε εάν η μέθοδος που γράφουμε θα πρέπει να μετακινηθεί στις συγκεκριμένες υποκλάσεις

# Interfaces, ξανά

- Όταν μια κλάση υλοποιεί ένα interface, θα πρέπει να παρέχει υλοποιήσεις σε **όλες** τις μεθόδους που δηλώνονται στο Interface
- Μπορεί να υπάρχουν **πολλές** μέθοδοι

```
interface KeyListener {
 public void keyPressed(KeyEvent e);
 public void keyReleased(KeyEvent e);
 public void keyTyped(KeyEvent e);
}
```

- Τι κάνουμε αν ενδιαφερόμαστε μόνο για **μερικές** από τις μεθόδους του interface και δεν θέλουμε η κλάση να είναι αφηρημένη?

# Adapter classes

- ΛΥΣΗ: χρησιμοποιούμε μια **adapter class**
- Μια **adapter class** υλοποιεί ένα interface και παρέχει κενά σώματα μεθόδων

```
class KeyAdapter implements KeyListener {
 public void keyPressed(KeyEvent e) { };
 public void keyReleased(KeyEvent e) { };
 public void keyTyped(KeyEvent e) { };
}
```

- Έτσι μπορούμε να επιλέξουμε ποιες από τις μεθόδους θα επικαλύψουμε (θα κάνουμε override) και στη συνέχεια να τις υλοποιήσουμε
- Δεν είναι μια κομψή λύση αλλά λειτουργεί
- Η Java παρέχει ένα πλήθος από έτοιμες κλάσεις adapter

<https://www.javatpoint.com/java-adapter-classes>

# Όροι

- **abstract μέθοδος** - μια μέθοδος που δηλώνεται αλλά δεν ορίζεται (δεν έχει σώμα)
- **abstract κλάση** - μια κλάση που είτε περιέχει τουλάχιστον μια αφηρημένη μέθοδο, ή απλά έχει δηλωθεί ως **abstract**
- **Instantiate** - δημιουργώ ένα στιγμιότυπο της κλάσης (δηλ. ένα αντικείμενο)
- **Interface** - μοιάζει με κλάση αλλά περιλαμβάνει μόνο αφηρημένες μεθόδους (και πιθανόν και σταθερές)
- **adapter κλάση** - μια κλάση που υλοποιεί ένα interface αλλά έχει μόνο κενά σώματα σε κάποιες μεθόδους

# Abstract classes vs Interfaces

- Main difference is methods of a Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behavior.
- Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
- Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, protected, etc..
- Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
- An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- A Java class can implement multiple interfaces but it can extend only one abstract class.
- Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
- In comparison with java abstract classes, java interfaces are slow as it requires extra indirection.

# Διαφορές Class vs Interface

| Class                                                                      | Interface                                                             |
|----------------------------------------------------------------------------|-----------------------------------------------------------------------|
| In class, you can instantiate variable and create an object.               | In an interface, you can't instantiate variable and create an object. |
| Class can contain concrete(with implementation) methods                    | The interface cannot contain concrete(with implementation) methods    |
| The access specifiers used with classes are private, protected and public. | In Interface only one specifier is used- Public.                      |