ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

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

interfaces
Abstract Classes

Αθ. Ανδρούτσος

Δημόσια Διεπαφή - API

- Δημόσια διεπαφή ή αλλιώς public API (Application Programming Interface) ή Open API μιας κλάσης είναι **το σύνολο των δημόσιων μεθόδων** της
- Είναι σημαντικό να μπορούμε να ορίζουμε το Public API ανεξάρτητα από την υλοποίηση
- Δηλαδή, να **διαχωρίζουμε (decouple) το Public API από την υλοποίηση** μιας και το ίδιο Public API μπορεί να υλοποιηθεί με διάφορους τρόπους

Interfaces (1)

- Ένα interface, στην Java, είναι μια δομή, όπως η κλάση, όπου όμως **δηλώνουμε μόνο τις επικεφαλίδες δημόσιων μεθόδων** (μέχρι και την Java 7) δηλαδή το public API χωρίς όμως υλοποίηση
- Οι μέθοδοι που δηλώνονται σε ένα interface είναι by default, **public και abstract (abstract** ονομάζονται οι μέθοδοι χωρίς υλοποίηση) ενώ επίσης ένα interface μπορεί να περιλαμβάνει **public static final** πεδία

Interfaces (2)

- Από την Java 8 και μετά υπάρχει και η δυνατότητα δήλωσης 1) default μεθόδων, δηλαδή μεθόδων που έχουν και υλοποίηση (για λόγους που θα αναλύσουμε αργότερα) καθώς και 2) static μεθόδων
- Επίσης, από την Java 9 και μετά υπάρχει και η δυνατότητα δήλωσης private και private static μεθόδων

Κλάσεις και Interfaces

- Λέμε ότι μία κλάση υλοποιεί (**implements**) ένα interface, όταν "κληρονομεί" από αυτό και υπερκαλύπτει τις μεθόδους του
- Οι κλάσεις που υλοποιούν (implement) ένα interface πρέπει υποχρεωτικά να υλοποιήσουν τις μεθόδους (το API) που ορίζει το interface
- Κατά αυτό τον τρόπο επιτυγχάνεται διαχωρισμός του Public API (interface) και του τρόπου υλοποίησής του (κλάση που το υλοποιεί)

Συμβόλαιο

- Επομένως ένα interface μπορεί να θεωρηθεί και ως ένα **Συμβόλαιο (contract)** για το public API ή αλλιώς τις υπηρεσίες που παρέχουμε προς τον client
- Η κλάση που το υλοποιεί, υλοποιεί ***το Public API που δηλώνεται στο interface***

interfaces και κλάσεις

- Ένα interface ορίζει ένα **τύπο** δεδομένων (όπως και μία κλάση)
- Όταν μία κλάση υλοποιεί ένα interface, υποδηλώνεται μία σχέση **IS-A** όπως στην κληρονομικότητα
- Επομένως, ένα interface μπορεί να χρησιμοποιηθεί ως *τύπος* για να δηλώνουμε μεταβλητές

Speakable Project

ISpeakable.java

```
1  package testbed.ch16.speakable;
2
3  public interface ISpeakable {
4      /**
5       * Makes a sound like speech.
6       */
7      void speak();
8  }
```

- Έστω ένα interface *ISpeakable*. Κατά σύμβαση, τα interface identifiers μπορούνε να ξεκινάνε με ***I* κεφαλαίο (uppercase)**, ενώ το όνομα μπορεί να είναι δηλωτικό της υπηρεσίας ή των υπηρεσιών που παρέχουν
- Δεδομένου ότι τα interfaces ορίζουν public APIs θα πρέπει οι υπηρεσίες τους να τεκμηριώνονται με doc comments

Interface methods are Public abstract

ISpeakable.java

```
1  package testbed.ch16.speakable;
2
3  public interface ISpeakable {
4      /**
5       * Makes a sound like speech.
6       */
7      public abstract void speak();
8  }
```

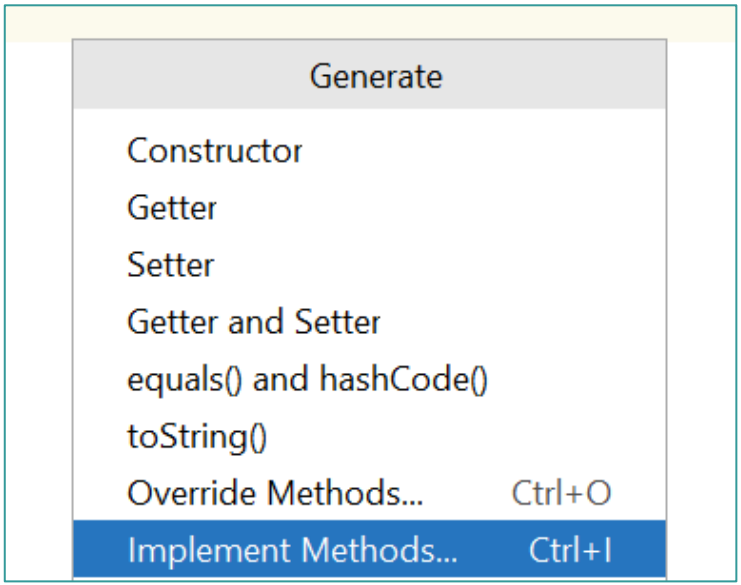
- Όπως βλέπουμε οι μέθοδοι των interfaces είναι public και abstract αλλά δεν χρειάζεται να το γράφουμε μιας και υπονοείται αυτόματα

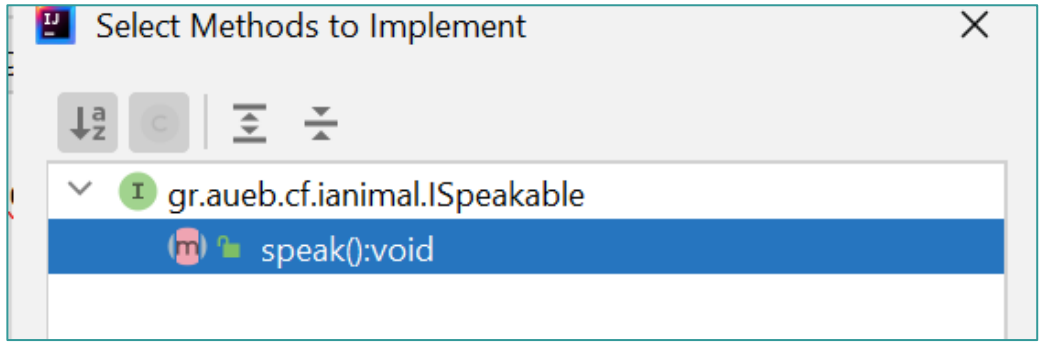
Cat implements ISpeakable

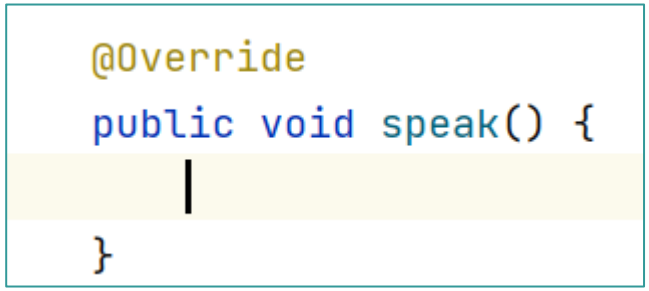

```java
package testbed.ch16.speakable;

public class Cat implements ISpeakable {
    private String name;

    public Cat() {}

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void speak() {
        System.out.println(name + " says hi");
    }
}
```

- Το ότι μία κλάση υλοποιεί ένα interface πρέπει να λέει κάτι σχετικά με **το τι ένας client μπορεί να κάνει** με ένα instance αυτής της κλάσης
- Εφόσον ***Cat IS-A ISpeakable***, σημαίνει πως τα instances της κλάσης Cat μπορούν να 'μιλήσουν' μέσω της υπηρεσίας speak που υποχρεωτικά υλοποιούν

IntelliJ







- Με Alt + Insert επιλέγουμε Implement Methods
- Στη συνέχεια επιλέγουμε τις μεθόδους που θέλουμε να κάνουμε implement και στη συνέχεια δημιουργούνται τα method stubs

Dog implements ISpeakable


```java
package testbed.ch16.speakable;

public class Dog implements ISpeakable {
    private String name;

    public Dog() {}

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void speak() {
        System.out.println(name + " says hello");
    }
}
```

- Το ίδιο μπορεί να ισχύει και με άλλες κλάσεις **χωρίς να χρειάζεται να είναι -οι κλάσεις που υλοποιούν ένα interface- μέρος κάποιας ιεραρχίας** κληρονομικότητας
- Δεν πρέπει επομένως να σκεφτόμαστε σε όρους κληρονομικότητας αλλά σε όρους λειτουργικότητας μιας και τα interfaces χρησιμοποιούνται για απόδοση λειτουργικότητας

Subtype Polymorphism (1)


```
1   package testbed.ch16.speakable;
2
3   public class Main {
4
5       public static void main(String[] args) {
6           ISpeakable alice = new Cat("Alice");
7           ISpeakable bob = new Dog("Bob");
8
9           alice.speak();
10          bob.speak();
11      }
12  }
```

- Τα interfaces **δεν είναι instantiable.** Ωστόσο εφόσον ***Cat IS-A ISpeakable***, όπως και στον πολυμορφισμό υποτύπων, έτσι και εδώ μπορεί η μεταβλητή να έχει τύπο του interface και η υλοποίηση να είναι οποιασδήποτε κλάσης υλοποιεί το interface

Subtype Polymorphism (2)


```java
package testbed.ch16.speakable;

public class Main {

    public static void main(String[] args) {
        ISpeakable alice = new Cat("Alice");
        ISpeakable bob = new Dog("Bob");

        alice.speak();
        bob.speak();
    }
}
```

- Αυτή η δυνατότητα είναι σημαντική γιατί μπορούμε να διαχωρίσουμε τον τύπο του interface από την υλοποίηση

Method Polymorphism


```
1   package testbed.ch16.speakable;
2
3   public class Main {
4
5       public static void main(String[] args) {
6           ISpeakable alice = new Cat("Alice");
7           ISpeakable bob = new Dog("Bob");
8
9           // alice.speak();
10          // bob.speak();
11
12          doSpeak(alice);
13          doSpeak(bob);
14      }
15
16      public static void doSpeak(ISpeakable speakable) {
17          speakable.speak();
18      }
19  }
```

- Η μέθοδος *doSpeak()* είναι πολυμορφική γιατί η τυπική παράμετρος speakable τύπου ISpeakable μπορεί να πάρει πολλές μορφές
- Μπορούμε δηλαδή να περάσουμε πραγματικές παραμέτρους οποιουδήποτε τύπου υλοποιεί το interface ISpeakable
- Όπως για παράδειγμα μπορούμε να περάσουμε instances τύπου Cat ή Dog ή και οποιουδήποτε τύπου στο μέλλον υλοποιήσει το ISpeakable

Polymorphism


```java
package testbed.ch16.speakable;

public class Main {

    public static void main(String[] args) {
        ISpeakable alice = new Cat("Alice");
        ISpeakable bob = new Dog("Bob");

        // alice.speak();
        // bob.speak();

        doSpeak(alice);
        doSpeak(bob);
    }

    public static void doSpeak(ISpeakable speakable) {
        speakable.speak();
    }
}
```

- Η doSpeak() είναι agnostic. Δεν γνωρίζει ποιο instance θα τις κάνουμε inject @runtime
- Επομένως δεν είναι στενά συνδεδεμένη με τον πραγματικό τύπο που της περνάμε ως παράμετρο (loosely coupled) και ως εκ τούτου η εφαρμογή μας είναι scalable (γιατί μπορούμε εύκολα να την επεκτείνουμε και με άλλες κλάσεις που υλοποιούν το ISpeakable), και testable (επειδή μπορούμε να την τεστάρουμε αυτόματα με διάφορα mock instances)

Interfaces και composition

- Ένα σημαντικό χαρακτηριστικό στη χρήση **interfaces και πολυμορφισμού βρίσκεται στο πλαίσιο του Composition**
- Έστω ότι έχουμε ένα χώρο όπου οι γάτες, και οι σκύλοι μαθαίνουν να μιλούν
- Ας δούμε το επόμενο παράδειγμα

Speaking School


```
1  package testbed.ch16.speakable;
2
3  public class SpeakingSchool {
4      private final Cat cat = new Cat();
5
6      public SpeakingSchool() {
7      }
8
9      public void learnToSpeak() {
10         cat.speak();
11     }
12 }
```

- Παρατηρούμε ότι το *SpeakingSchool* περιέχει ένα *Cat* private instance **και αρχικοποιεί το δικό του instance με new()**
- Αυτό κάνει το SpeakingSchool **tightly coupling** με το *Cat* και περιορίζει σημαντικά την δυνατότητά του
- Αν μία γάτα χρειάζεται εκπαίδευση το SpeakingSchool είναι εδώ
- Αν όμως ένα Dog χρειάζεται εκπαίδευση;

Coupling


```
1  package testbed.ch16.speakable;
2
3  public class SpeakingSchool {
4      private final Cat cat = new Cat();
5
6      public SpeakingSchool() {
7      }
8
9      public void learnToSpeak() {
10         cat.speak();
11     }
12 }
```

- Το **tightly coupling** , δηλαδή το ότι το *SpeakingSchool* συνδέεται με μία συγκεκριμένη υλοποίηση (Cat) --που είναι γνωστή @compile time– δεν μπορεί ουσιαστικά να τεσταριστεί με διάφορα mock objects

Από την άλλη πλευρά χρειαζόμαστε ένα βαθμό coupling μιας και οι κλάσεις πρέπει να συνεργάζονται όπως εδώ μέσα από το composition

Συνεργασία κλάσεων - Dependencies

- Μία εφαρμογή είναι ένα σύνολο αλληλεξαρτώμενων instances κλάσεων. Δεν μπορεί μία κλάση να τα κάνει όλα μόνη της, οπότε χρειάζεται αλληλεξάρτηση
- Το SpeakingSchool για παράδειγμα θα ήθελε ιδανικά να συνεργάζεται με instances κλάσεων όπως Cat, Dog και γενικά ISpeakable. Στην πραγματικότητα αυτά τα instances, όπως το Cat, αποτελούν εξαρτήσεις (dependencies) για την SpeakingSchool. Η λειτουργία της εξαρτάται από τη συνεργασία της με αυτά τα instances

Coupling (1)

- Επομένως χρειαζόμαστε αυτή τη μορφή συνεργασίας και σύνδεσης (coupling) μεταξύ κλάσεων και instances κλάσεων
- Από την άλλη πλευρά θα θέλαμε η σύνδεση αυτή να είναι χαλαρή (**loosely coupling**) και όχι ισχυρή (**tightly coupling**) γιατί κάτι τέτοιο δυσχεραίνει τη διαδικασία ελέγχου μιας και δεν μπορούμε να τεστάρουμε ανεξάρτητα μία κλάση που έχει ισχυρές εξαρτήσεις

Coupling (2)

- Επομένως χρειάζεται μία ισορροπία μεταξύ διασύνδεσης κλάσεων που είναι απαραίτητη για τη συνεργασία στο πλαίσιο εφαρμογών και από την άλλη μία μορφή ανεξαρτησίας που είναι απαραίτητη για τον έλεγχο των ανεξάρτητων μονάδων (μεθόδων και κλάσεων) του λογισμικού μας
- Αυτό μπορεί να επιτευχθεί με τη χρήση interfaces αντί κλάσεων στο πλαίσιο του Composition, όπως θα δούμε στη συνέχεια

Composition με interface (1)


```java
package testbed.ch16.speakable;

public class GenericSpeakingSchool {
    private final ISpeakable speakable;

    public GenericSpeakingSchool(ISpeakable speakable) {
        this.speakable = speakable;
    }

    public void learnToSpeak() {
        speakable.speak();
    }
}
```

- Το GenericSpeakingSchool περιέχει ένα ISpeakable instance. Ο Constructor αντί να κάνει new με συγκεκριμένο τύπο (π.χ. cat) του δίνουμε (inject) ένα ISpeakable instance @runtime

Dependency Injection – Inversion of Control

- Αυτό είναι γνωστό ως DI (Dependency Injection) ή αλλιώς IoC (Inversion of Control). Εδώ υλοποιείται με Constructor Injection.
- Το σημαντικό είναι ότι το GenericSpeakingSchool είναι loosely coupled με το αντικείμενο που περιέχει μιας και δεν το δημιουργεί, αλλά του δίνεται @runtime
- Το **Dependency injection** επιτυγχάνεται με τη χρήση **interfaces ως instances κλάσεων** και τη χρήση μεθόδων όπως constructors για να κάνουμε inject instances κλάσεων που υλοποιούν το interface

Composition με interface


```java
package testbed.ch16.speakable;

public class GenericSpeakingSchool {
    private final ISpeakable speakable;

    public GenericSpeakingSchool(ISpeakable speakable) {
        this.speakable = speakable;
    }

    public void learnToSpeak() {
        speakable.speak();
    }
}
```

- Το GenericSpeakingSchool είναι agnostic του τι αντικείμενο περιέχει, από τη στιγμή που περιέχει ένα interface. Το IoC μπορεί να υλοποιηθεί με οποιαδήποτε υλοποίηση ενός ISpeakable κάτι που διαχωρίζει το dependency από την υλοποίηση (loosely coupled)
- Η κλάση GenericSpeakingSchool είναι testable και μπορεί να τεσταριστεί με ένα mock object framework όπως το Mockito

Wiring


```
1   package testbed.ch16.speakable;
2
3   public class Main {
4
5       public static void main(String[] args) {
6           ISpeakable whiteCat = new Cat("White Cat");
7           ISpeakable blackDog = new Dog("Black Dog");
8
9           GenericSpeakingSchool aliceSpeakingSchool = new GenericSpeakingSchool(whiteCat);
10          GenericSpeakingSchool bobSpeakingSchool = new GenericSpeakingSchool(blackDog);
11      }
12  }
```

- Πως μπορούμε να συνδέσουμε (wiring) ένα Cat ή ένα Dog με ένα GenericSpeakingSchool;
- Μπορούμε να το κάνουμε **@construction time.** Κάνουμε inject το whiteCat και το blackDog μέσω του GenericSpeakingSchool Constructor (γραμμές 9 και 10)

Marker interfaces

- Η Java δίνει τη δυνατότητα να ορίζουμε κενά interfaces, χωρίς καμία μέθοδο (**marker interfaces**) για να κάνουμε mark μία class που υλοποιεί το interface ως τέτοια
- Συνακόλουθα μπορούμε να κάνουμε catch errors @compile time, αν για παράδειγμα περνάμε σε μια πολυμορφική μέθοδο ένα object που δεν υλοποιεί το marker interface

Marker Interface


```java
package testbed.ch16.marker;

/**
 * {@link Item} is a marker interface
 *
 * @author a8ana
 */
public interface Item {
}
```

- Ορίζουμε ένα interface χωρίς μεθόδους

Book / CompactDisk


```java
package testbed.ch16.marker;

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

    public Book() {}

    public Book(long id, String isbn, String author, String title) {...}

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }
    public String getIsbn() { return isbn; }
    public void setIsbn(String isbn) { this.isbn = isbn; }
    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

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

```java
package testbed.ch16.marker;

public class CompactDisk implements Item {
    private long id;
    private String title;

    public CompactDisk() {
    }

    public CompactDisk(long id, String title) {...}
    public long getId() { return id; }
    public void setId(long id) { this.id = id; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

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

- Ορίζουμε δύο κλάσεις Book και CompactDisk που κάνουν implements το Item, ώστε να μπορούμε να τις χρησιμοποιήσουμε στο πλαίσιο του πολυμορφισμού

Polymorphism / Deliver


```java
package gr.aueb.cf.ch16.marker;

public class Main {

    public static void main(String[] args) {
        Item book = new Book(1, "123", "Th.", "Java I");
        Item cd = new CompactDisk(1, "Prince");

        deliver(book);
        deliver(cd);
    }

    public static void deliver(Item item) {
        if (item instanceof Book) {
            System.out.println("Book delivered");
        } else if (item instanceof CompactDisk) {
            System.out.println("CD delivered");
        } else {
            throw new AssertionError(item);
        }
    }
}
```

- Η deliver είναι πολυμορφική και μπορεί να λάβει ως πραγματική παράμετρο και Book και CompactDisk αφού υλοποιούν το Item
- Οτιδήποτε άλλο που δεν είναι Item δημιουργεί compile-time error

Functional Interfaces (1)

- Ένα Interface με μία μόνο abstract μέθοδο ονομάζεται ***Functional Interface***
- Μπορούμε να χρησιμοποιήσουμε το annotation **@FunctionalInterface** ώστε να κάνουμε tag το interface ως Functional και να ελέγξει ο μεταγλωττιστής αν πράγματι το interface έχει μία μέθοδο και να απαγορεύει στο μέλλον να προστεθούν άλλες abstract μέθοδοι
- Τα tags λειτουργούν ως μεταδεδομένα (metadata) και η χρήση τους είναι προαιρετική αλλά συνίστανται

Functional Interfaces (2)

- Τα Functional interfaces λειτουργούν ως functions (η Java δεν έχει native functions) και η Java μας επιτρέπει να χρησιμοποιούμε *Lambda Expressions* ως callbacks
- Είναι ανάλογο των arrow functions της JavaScript

Εξέλιξη Interfaces στην Java

- Μέχρι την **έκδοση 7 της Java**, ένα **interface** μπορούσε να έχει μόνο static final πεδία και abstract μεθόδους
- Από την **έκδοση 8 της Java** τα interfaces μπορούσαν να περιέχουν και static και default μεθόδους δηλαδή μεθόδους με υλοποίηση
- Επομένως από την έκδοση 8 και μετά τα interfaces μπορούσαν να έχουν:
  - Πεδία σταθερά (final static)
  - Μεθόδους Abstract / Default / Static
- Από την έκδοση 9 της Java τα **Interfaces** μπορούν να περιέχουν και **private** και **private static** μεθόδους

Default Μέθοδοι (1)

- Ο λόγος που χρειαζόμαστε τις default μεθόδους στα interfaces είναι για λόγους συντήρησης του κώδικα.
- Καθώς περνάει ο καιρός και όλο και περισσότερες εφαρμογές υλοποιούν ένα *interface*, αν θα θέλαμε να προσθέσουμε μια νέα abstract μέθοδο στο interface τότε γίνονται break οι υλοποιήσεις όλων των εφαρμογών που υλοποιούν το Interface οι οποίες θα πρέπει τώρα να υλοποιήσουν και τη νέα μέθοδο

Default Μέθοδοι (2)

- Μπορώντας όμως να γράψουμε μέσα στο interface **default μεθόδους**, ***δηλαδή μεθόδους που παρέχουν και υλοποίηση***, οι κλάσεις που υλοποιούν το **interface**, μπορούν να λειτουργήσουν χωρίς αλλαγές και μπορούν να χρησιμοποιήσουν τη νέα default μέθοδο την οποία αν θέλουν μπορούν να την υπερκαλύψουν (override)

Static μέθοδοι σε Interfaces

- Το ίδιο ισχύει και για τις **static** μεθόδους οι οποίες κάνουν την ίδια δουλειά όπως και οι **default** μέθοδοι **μόνο που δεν μπορούν να γίνουν override**
- Αυτό το παρέχει η Java για να **μην μπορούν οι υποκλάσεις** να παρέχουν επικαλυμμένες (overridden) μεθόδους με φτωχή υλοποίηση

Private μέθοδοι στα Interfaces

- Στην Java 9 έχουν προστεθεί **private μέθοδοι (και private static)** στα Interfaces
- Αυτό έχει γίνει ώστε να μπορούν οι default μέθοδοι να σπάσουν σε υποπρογράμματα που θα υλοποιούνται σε private μεθόδους (όσο δεν υπήρχαν private μέθοδοι οι default μέθοδοι μπορεί να γίνονταν πολύ μεγάλες και πολύπλοκες)
- Αντίστοιχα οι static μέθοδοι μπορούν να ‘σπάσουν’ σε private static μεθόδους

Default / Static methods


```java
package testbed.ch16.defaults;

public interface IWelcome {
    1 implementation
    void saySomething(String message);

    1 override
    default void sayHelloCodingFactory() {
        System.out.print("Hello ");
        sayCodingFactory();
    }

    private void sayCodingFactory() {
        System.out.println("Coding Factory");
    }

    static void sayHelloCoding() {
        System.out.println("Hello");
        sayCoding();
    }

    private static void sayCoding() {
        System.out.println("Coding");
    }
}
```

- Έστω ένα interface με διάφορους τύπους μεθόδων
  - Abstract
  - Default
  - Private
  - Static
  - Private static

CodingFactory implements IWelcome


```java
package testbed.ch16.defaults;

public class CodingFactory implements IWelcome {
    @Override
    public void saySomething(String message) {
        System.out.println("Please read the message: " + message);
    }

    @Override
    public void sayHelloCodingFactory() {
        IWelcome.super.sayHelloCodingFactory();
        System.out.println("\u2764".repeat(5));   // Red Heart
    }
}
```

- Μία κλάση πρέπει να κάνει override τις abstract μεθόδους και προαιρετικά τις default

Main


```java
package testbed.ch16.defaults;

public class Main {

    public static void main(String[] args) {
        IWelcome welcomeCf = new CodingFactory();
        welcomeCf.sayHelloCodingFactory();
    }
}
```

Run: testbed.ch16.defaults.Main

```
"C:\Program Files\Amazon Corretto\jdk11.0.10_9\bin\j
Hello Coding Factory
❤️❤️❤️❤️❤️

Process finished with exit code 0
```

Interfaces – multiple inheritance

- Τα interfaces επιτρέπουν ***πολλαπλή κληρονομικότητα,*** μπορεί δηλαδή **ένα interface να κάνει extends περισσότερα από ένα interfaces** επιτρέποντας έτσι σύνθεση interfaces εν αντιθέσει με τις κλάσεις όπου επιτρέπεται μόνο απλή / μονή κληρονομικότητα
- Επίσης, **μπορεί μια κλάση να υλοποιεί πολλά interfaces**

Πολλαπλή κληρονομικότητα (1)


```java
package testbed.ch16.inh;

@FunctionalInterface
public interface ISpeakable {
    1 implementation
    void speak();
}
```

```java
package testbed.ch16.inh;

@FunctionalInterface
public interface IReadable {
    1 implementation
    void read();
}
```

- Τα interfaces, εν αντιθέσει με τις κλάσεις, επιτρέπουν πολλαπλή κληρονομικότητα.
- Έστω τα παραπάνω interfaces, ISpeakable και IReadable

Πολλαπλή κληρονομικότητα (2)


```
1   package testbed.ch16.inh;
2
3   public interface ITalking extends ISpeakable, IReadable {
4   }
```

- Σε αντίθεση με τις κλάσεις που δεν παρέχουν πολλαπλή κληρονομικότητα, τα interfaces μπορούν να κάνουν extends από περισσότερα από ένα interfaces

Πολλαπλή κληρονομικότητα (3)


```java
package testbed.ch16.inh;

public class TalkingBook implements ITalking {
    @Override
    public void read() {
        System.out.println("Can be read");
    }

    @Override
    public void speak() {
        System.out.println("Can speak");
    }
}
```

- Το TalkingBook κάνει implements το ITalking

Abstract Classes

- Οι abstract κλάσεις παρέχουν μία base class την οποία άλλες κλάσεις μπορούν να κάνουν extend
- Οι abstract classes περιέχουν:
  - **Implemented methods.** Μεθόδους που είναι πλήρως υλοποιημένες και οι κλάσεις που κληρονομούν μπορούν να χρησιμοποιήσουν ή να κάνουν override
  - **Abstract methods.** Μεθόδους που δεν είναι υλοποιημένες και οι κλάσεις που κληρονομούν πρέπει να υλοποιήσουν (override)
- Οι abstract κλάσεις ορίζονται με το **abstract keyword** (π.χ. *public abstract class AbstractStudent*)

Abstract μέθοδοι και κλάσεις

- Οι Abstract κλάσεις μπορεί να περιέχουν ***μηδέν, μία ή περισσότερες abstract μεθόδους*** (μπορεί δηλαδή οι abstract κλάσεις να μην περιέχουν και καμία abstract μέθοδο)
- Μπορούν να παρέχουν δημιουργούς **αλλά δεν μπορούν να κάνουν instantiate αντικείμενα.** Γίνονται instantiate μόνο μέσω των υποκλάσεων
- Μπορούν να έχουν πεδία (fields) όπως και οι κανονικές κλάσεις

Abstract Κλάσεις

- Οι Abstract κλάσεις έχουν δημιουργηθεί για να κληρονομούνται.
- Επομένως **ΠΡΕΠΕΙ να γίνουν subclassed**
- Οι concrete κλάσεις μπορούν να κάνουν **extends** μία abstract κλάση

Abstract Animal


```java
package testbed.ch16.abstra;

public abstract class Animal {
    private long id;
    private String name;

    public Animal() {}

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @Override
    public String toString() { return "Id: " + id + " Name: " + name; }

    public abstract void speak();

    public void eat() {
        System.out.println(this.getClass().getSimpleName() + " is eating");
    }
}
```

- Ορίζουμε μια Abstract κλάση με το abstract keyword
- Περιλαμβάνει, **όσα περιλαμβάνει και μία κανονική κλάση και επιπλέον μπορεί (χωρίς να είναι απαραίτητο) να περιλαμβάνει abstract μεθόδους** όπως η speak() με το keyword abstract

Cat extends AbstractAnimal


```java
package testbed.ch16.abstra;

public class Cat extends Animal {

    // Should be overridden
    @Override
    public void speak() {
        System.out.println("Νιάου");
    }

    // May ne overridden
    @Override
    public void eat() {
        super.eat();
        System.out.println("... She ate all her food!");
    }
}
```

- Η κλάση Cat κάνει extends όπως θα έκανε και σε μία concrete κλάση.
- Εδώ ΥΠΟΧΡΕΩΤΙΚΑ θα πρέπει να κάνει override την abstract μέθοδο (speak) της Abstract class και προαιρετικά την overridable eat

Abstract κλάσεις vs interfaces

- Η Java παρέχει δύο μηχανισμούς για να ορίζουμε τύπους που επιτρέπουν πολλαπλές υλοποιήσεις: τα interfaces και τις abstract classes
- Μετά την εισαγωγή στα interfaces (στην Java 8, JLS 9.4.3) των default methods και οι δύο μηχανισμοί επιτρέπουν τον ορισμό implementation για instance methods

Abstract κλάσεις vs interfaces (1)

- Μία σημαντική διαφορά είναι ότι στις Abstract Classes επιτρέπεται μόνο *μονή κληρονομικότητα*
- Ενώ στα interfaces επιτρέπεται *μία κλάση να υλοποιεί περισσότερα από ένα* interfaces

Abstract κλάσεις vs interfaces (2)

- Επίσης, για να υλοποιήσει μία κλάση ένα interface **δεν χρειάζεται να βρίσκεται σε μία ιεραρχία κληρονομικότητας**. Το μόνο που χρειάζεται είναι ένα implements clause και να υλοποιήσει τις απαιτούμενες μεθόδους
- Επομένως μπορεί μία κλάση πολύ εύκολα, εκ των υστέρων να υλοποιήσει ένα interface ή περισσότερα interfaces, κάτι που δεν μπορεί να συμβεί στην κληρονομικότητα, όπου από την αρχή θα πρέπει μία κλάση να κληρονομήσει από μία abstract κλάση και δεν μπορεί στη συνέχεια να κληρονομήσει και από άλλη abstract κλάση

Mixins

- Αφού μπορεί μία κλάση εκ των υστέρων να υλοποιήσει ένα interface, μπορούμε να ορίζουμε ***mixin interfaces***, δηλαδή interfaces που μπορεί μία κλάση να υλοποιήσει επιπλέον του βασικού της τύπου
- Για παράδειγμα το *interface Comparable* είναι ένα *mixin* interface που επιτρέπει σε μια κλάση να δηλώσει ότι τα instances της μπορούν να ταξινομηθούν
- Οι abstract κλάσεις δεν μπορούν να ορίσουν mixins για τον ίδιο λόγο που δεν μπορεί μία κλάση να κάνει extends εκ των υστέρων μία abstract κλάση

Interfaces και Object class

- Τα interfaces δηλώνουν έμμεσα ως public abstract τις μεθόδους της Object class, όπως η equals, ή η hashCode, αλλά δεν μπορούν να ορίσουν default methods για αυτές
- Τα interfaces δεν κληρονομούν από την Object class αλλά (αν δεν έχουν άμεσα superinterfaces) έμμεσα ορίζουν abstract methods με την ίδια υπογραφή για κάθε public μέθοδο της Object class
- Ο λόγος είναι πως επειδή μπορούμε να χρησιμοποιήσουμε interfaces για type declarations θα θέλαμε να μπορούμε να χρησιμοποιούμε τις μεθόδους της Object class, όπως έχουν γίνει override στην concrete class που υλοποιεί το interface

Abstract Κλάσεις – Skeletal Implementations

- Εφόσον προτιμώνται τα interfaces σε σχέση με τα Abstract classes, για ποιο λόγο έχουμε Abstract κλάσεις και πως χρησιμοποιούνται;
- Οι Abstract κλάσεις **συνίσταται να χρησιμοποιούνται σε συνδυασμό με interfaces** ώστε να παρέχουν *Skeletal Implementations* δηλαδή ***υλοποίηση των abstract μεθόδων των interfaces από τα οποία κληρονομούν ώστε να μην χρειάζεται οι κληρονομούμενες κλάσεις να υλοποιούν τις μεθόδους του interface***

Skeletal Implementations

- Κατά σύμβαση τα skeletal implementation classes ονομάζονται **Abstract*Interface***, όπου Interface είναι το όνομα του interface που υλοποιούν
- Η ομορφιά των skeletal implementations είναι ότι παρέχουν όλη την implementation assistance προς τις κλάσεις που θέλουν να υλοποιήσουν ένα interface, χωρίς τα μειονεκτήματα και τους περιορισμούς που υπάρχουν όταν ορίζονται ως type definitions
- Αν μια κλάση δεν θέλει να κάνει extends ένα abstract class μπορεί πάντα να υλοποιήσει μόνη της το interface

IRectangle


```java
package gr.aueb.cf.ch16.skeletal;

public interface IRectangle {

    double getArea();
    double getPerimeter();
    boolean isSquare();
}
```

- Έστω το interface IRectangle. Περιέχει τρεις μεθόδους, κοινές για όλα τα ορθογώνια παραλληλόγραμμα

AbstractRectangle


```java
package gr.aueb.cf.ch16.skeletal;

public class AbstractRectangle implements IRectangle {
    private double width;
    private double height;

    @Override
    public double getArea() {
        return width * height;
    }

    @Override
    public double getPerimeter() {
        return 2 * (width + height);
    }

    @Override
    public boolean isSquare() {
        return width == height;
    }
}
```

- Το AbstractRectangle παρέχει ένα skeletal implementation του IRectangle

Square


```java
package gr.aueb.cf.ch16.skeletal;

public class Square extends AbstractRectangle {

}
```

- Δεν χρειάζεται να υλοποιήσουμε το Square αφού το AbstractRectangle παρέχει μία default υλοποίηση

Isquare (1)

- Μία παραλλαγή του προηγούμενου παραδείγματος θα ήταν αν πριν υλοποιήσουμε το Square είχαμε ένα ISquare που έκανε extends το IRectangle και πιθανά προσέθετε και μία ή περισσότερες πιο ειδικές μεθόδους που θα πρέπει να παρέχει το API του ISquare

ISquare (2)


```
1   package gr.aueb.cf.ch16.skeletal;
2
3   public interface ISquare extends IRectangle {
4
5       double getDiagonal();
6   }
```

- Το interface ISquare κάνει extends ένα generic interface, το IRectangle και προσθέτει επιπλέον μεθόδους

Square


```java
package gr.aueb.cf.ch16.skeletal;

public class Square extends AbstractRectangle implements ISquare {

    @Override
    public double getDiagonal() {
        return Math.sqrt(Math.pow(getWidth(), 2) + Math.pow(getHeight(), 2));
    }
}
```

- Το Square κάνει extends το generic implementation του AbstractRectangle και κάνει implements το ISquare που είναι το πιο ειδικό μέρος του Square

Abstract Classes – tag field

- Σε κάποιες περιπτώσεις μπορεί να έχουμε classes που περιέχουν παρόμοια αλλά διακριτά instances τα οποία διαχωρίζονται με βάση ένα tag field

Tagged classes (1)


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

public class Shape {
    private enum TYPE { RECTANGLE, CIRCLE }

    // Tag field
    private final TYPE type;

    // Fields for Rectangle
    private double width;
    private double height;

    // Fields for Circle
    private double radius;

    // Constructor for Rectangle
    public Shape(double width, double height) {
        type = TYPE.CIRCLE;
        this.width = width;
        this.height = height;
    }
```

- Ορίζουμε την tagged κλάση
- Η δήλωση *enum* έχει σύνταξη παρόμοια με της κλάσης. Βοηθάει όταν έχουμε πολλές σταθερές, αντί για final
- Με βάση το enum ορίζουμε ένα tag field καθώς και πεδία για κάθε περίπτωση (flavor)

Tagged classes (2)


```
23      // Constructor for Circle
24      public Shape(double radius) {
25          type = TYPE.RECTANGLE;
26          this.radius = radius;
27      }
28
29
30      public double area() {
31          switch (type) {
32              case CIRCLE:
33                  return radius * radius + Math.PI;
34              case RECTANGLE:
35                  return width * height;
36              default:
37                  throw new AssertionError(type);
38          }
39      }
40  }
```

- Ορίζουμε ένα constructor για κάθε περίπτωση
- Και σε κοινές μεθόδους έχουμε switch για κάθε περίπτωση

Tagged classes (3)

- Οι tagged classes έχουν πολλά προβλήματα. Boilerplate code με enum, tag field, switch. Χαμηλό readability μιας και όλα μαζί τα πεδία και οι constructors είναι μέσα σε μία κλάση. Μεγάλο memory footprint. Τα πεδία δεν μπορεί να γίνουν final εκτός αν οι constructors αρχικοποιούν άσχετα πεδία. Δεν μπορούμε να προσθέσουμε flavors παρά μόνο αν αλλάξουμε το source file και να θυμηθούμε και τη switch. Το data type δεν μας λέει ποιο flavor έχει ένα instance
- Με λίγα λόγια, τα tagged class είναι error-prone, verbose και μη-αποτελεσματικά

Abstract αντί Tagged classes

- Η λύση αντί για tagged classes θα ήταν να ορίσουμε μία Abstract class που να περιέχει τις κοινές μεθόδους και τα κοινά πεδία αν υπάρχουν που δεν είναι tag-dependent.
- Τις μεθόδους αυτές θα τις κάνουμε override στην subclass

Abstract Shape


```
1   package gr.aueb.cf.abstractshapenottagged;
2
3   public abstract class Shape {
4       public abstract double area();
5   }
```

- Το abstract shape μπορεί να έχει γενικά όλα τα κοινά πεδία και μεθόδους των υποκλάσεων (flavors)

Circle extends Shape


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

public class Circle extends Shape {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius;
    }
}
```

- Η κλάση Circle κάνει extends το Shape κάνοντας override την area()

Rectangle extends Shape

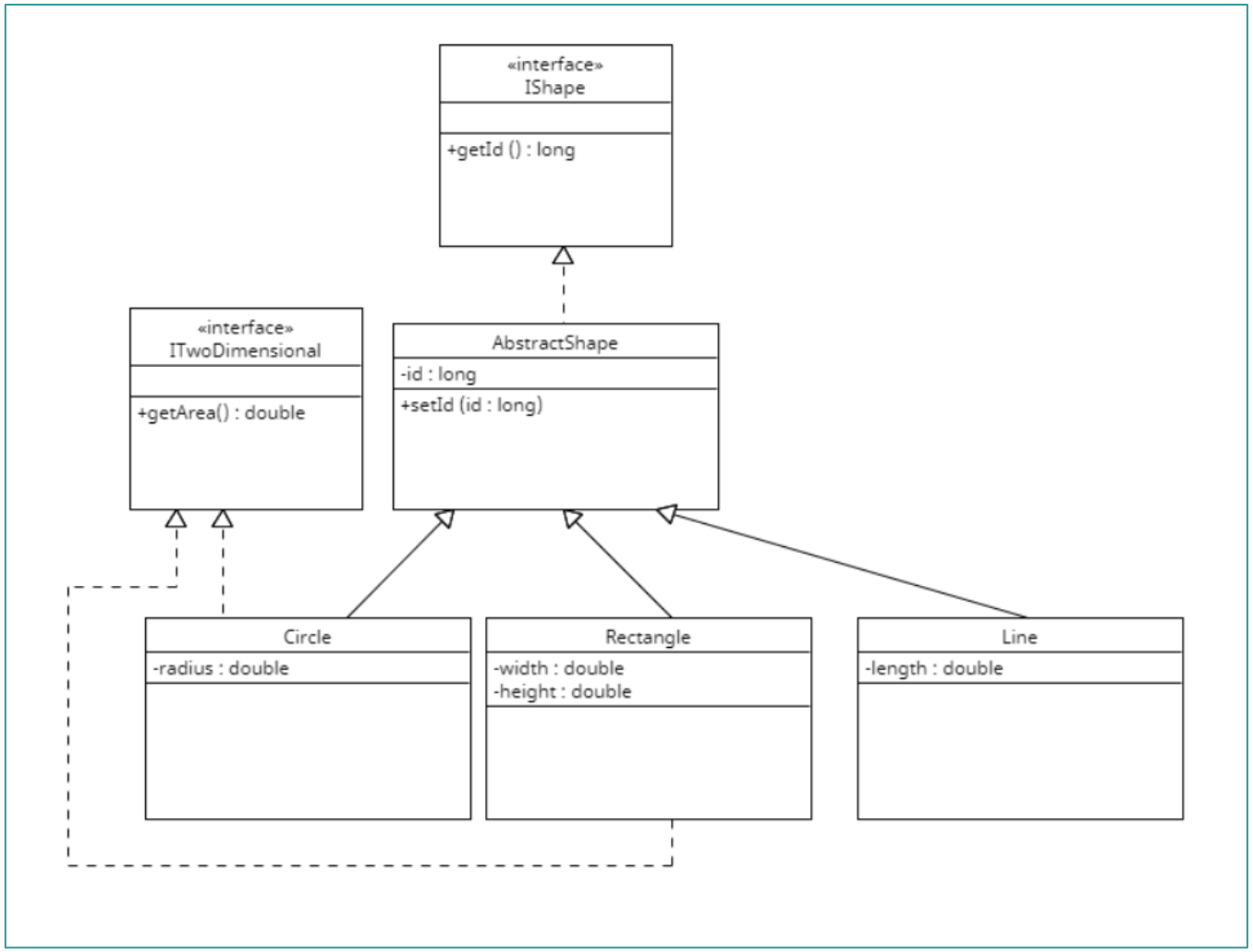

```java
package gr.aueb.cf.abstractshapenottagged;

public class Rectangle extends Shape {
    private final double width;
    private final double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}
```

- Η Rectangle θέτει τα δικά της πεδία όπως και η Circle και επαναορίσει την area()

Ασκήσεις Shapes

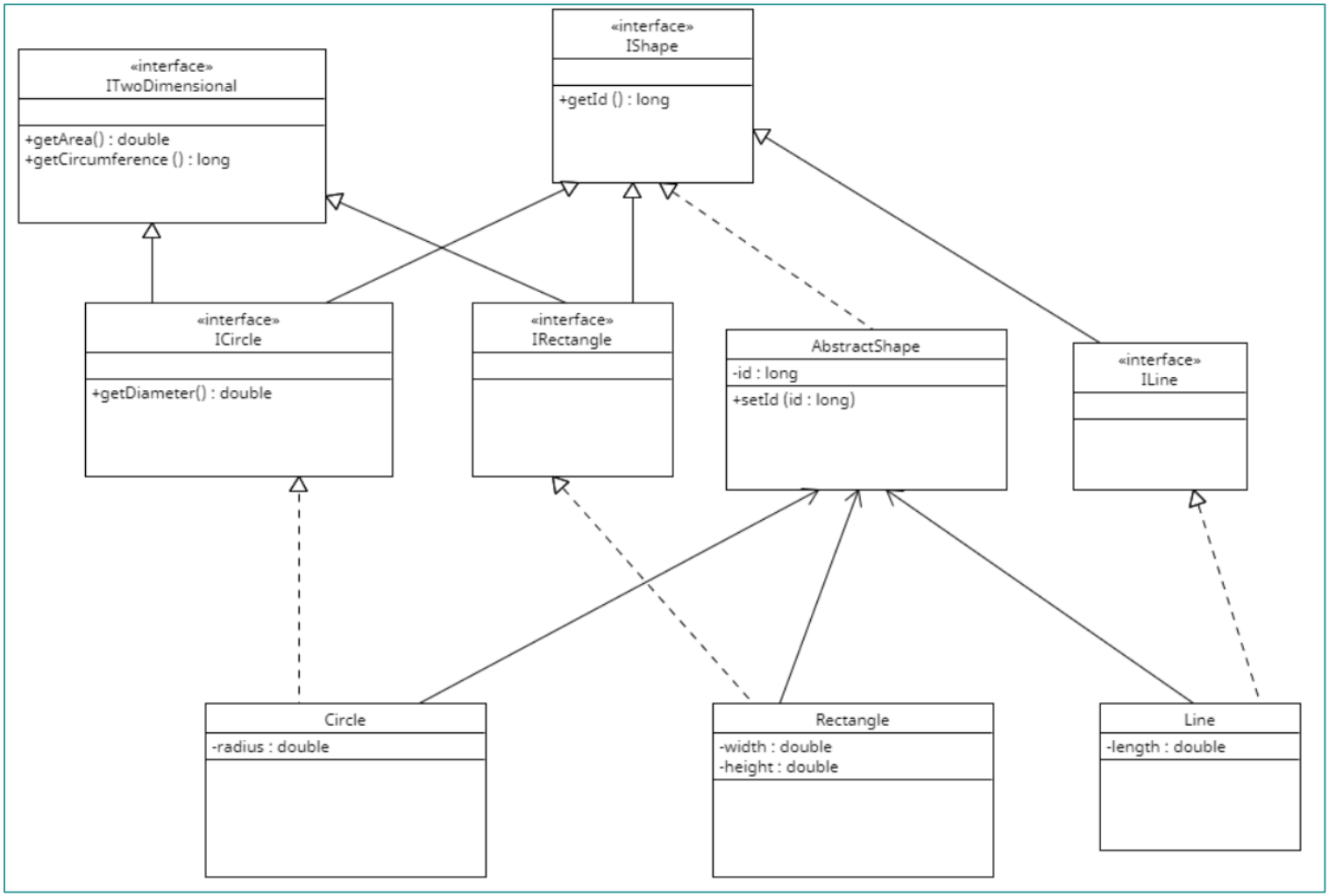
- Έστω ότι θέλουμε να δημιουργήσουμε κλάσεις για διάφορα σχήματα όπως *Line*, *Circle* και *Rectangle*
- Προσπαθήστε να υλοποιήσετε τις παρακάτω ασκήσεις

Ασκήσεις (1)



- Υλοποιήστε το UML σχήμα

Ασκήσεις (2)



- Υλοποιήστε το UML σχήμα