# ΔΙΑΠΡΟΣΩΠΕΙΕΣ

# Αφαιρετικές Μέθοδοι

ΕΠΛ233

- Στα προηγούμενα παραδείγματα, οι μέθοδοι της κλάσεως-κληροδότη **Instrument** είναι "πλασματικές", με την έννοια ότι ο ορισμός τους γίνεται για τον καθορισμό μιας *διαπροσωπείας* κοινής σε όλες τις κλάσεις-κληρονόμους της (common interface).
- Η Java μας επιτρέπει να δηλώνουμε ρητά ορισμένες μεθόδους μιας κλάσης σαν **αφαιρετικές** ώστε:
  - Να έχουμε τη δυνατότητα να προσδιορίζουμε την διαπροσωπεία της κλάσης που τις περιέχει.
  - Να αποτρέπεται η εκ παραδρομής κλήση τους από άλλες μεθόδους.
- Ο προσδιορισμός μιας μεθόδου ως αφαιρετικής γίνεται ως εξής:

```
abstract void X();
```

# Αφαιρετικές Κλάσεις

ΕΠΛ233

- Μια κλάση η οποία περιλαμβάνει έστω και μια «αφαιρετική» μέθοδο, καθίσταται επίσης αφαιρετική και πρέπει να δηλωθεί ως αφαιρετική.
- Τι συμβαίνει αν προσπαθήσουμε να δημιουργήσουμε ένα αντικείμενο από κάποια αφαιρετική κλάση;
  - Εφόσον μια κλάση κληρονομεί από μια *αφαιρετική κλάση*, και θέλουμε να δημιουργήσουμε αντικείμενα γι' αυτήν, θα πρέπει η κλάση μας να περιέχει τους ορισμούς για **όλες** τις αφαιρετικές μεθόδους της κλάσεως-βάσης. Διαφορετικά ο μεταφραστής επιβάλλει να προσδιορίσουμε την κλάση μας σαν αφαιρετική.
  - Μπορούμε τέλος να δηλώσουμε μια κλάση σαν αφαιρετική, **χωρίς** ωστόσο η κλάση αυτή να περικλείει αφαιρετικές μεθόδους. *Γιατί να θέλουμε κάτι τέτοιο;* Για να αποκλείσουμε τη δημιουργία αντικειμένων αυτής της κλάσης.

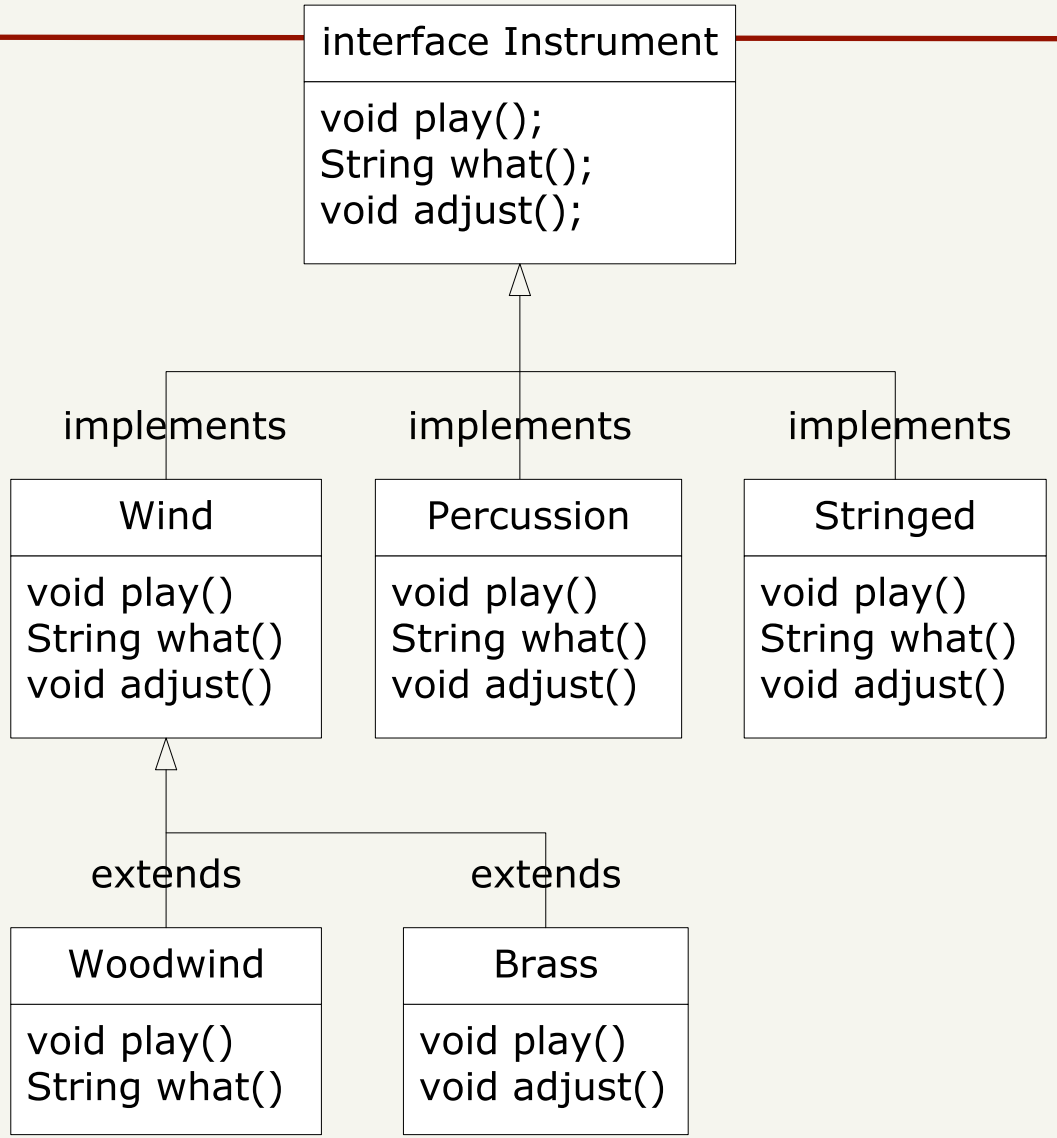
# Διαπροσωπείες-Interfaces

ΕΠΛ233

- Η Java επεκτείνει τις ιδέες της αφαιρετικότητας μεθόδων και κλάσεων που καθορίζονται με το **abstract,** μέσω του **interface.** Με το **interface** εισάγεται η έννοια μιας "καθαρής" αφηρημένης κλάσης.
- Επιτρέπεται έτσι στον προγραμματιστή να καθορίσει τη *μορφή* μιας κλάσης: ονόματα μεθόδων, καταλόγους παραμέτρων, τύπους επιστροφής, αλλά **όχι** σώματα μεθόδων.
- Επίσης, επιτρέπεται ο καθορισμός πεδίων δεδομένων, τα οποία ωστόσο προκαθορίζονται υποχρεωτικά ως **στατικά (static) και τελικά (final).**
- Για την δήλωση διαπροσωπειών, χρησιμοποιούμε τη λέξη κλειδί **interface** στη θέση της λέξης-κλειδί **class.**

# Παράδειγμα

ΕΠΛ233



# Παράδειγμα

ΕΠΛ233

```
interface Instrument {
    int i = 5; // static & final
    void play();
    String what();
    void adjust();
}
class Wind implements Instrument {
    public void play() { System.out.println("Wind.play()"); }
    public String what() { return "Wind"; }
    public void adjust() { }
}
class Percussion implements Instrument {
  public void play() {     System.out.println("Percussion.play()");  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}
```

# Παράδειγμα (συνέχεια)

ΕΠΛ233

```
 class Stringed implements Instrument {
  public void play() {    System.out.println("Stringed.play()");   }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {   System.out.println("Brass.play()");   }
  public void adjust() { System.out.println("Brass.adjust()");  }
}
class Woodwind extends Wind {
  public void play() {  System.out.println("Woodwind.play()");  }
  public String what() { return "Woodwind"; }
}
```

# Παράδειγμα (συνέχεια)



Doesn't care about type, so new types added to the system still work right

```
public class Music5 {
      static void tune(Instrument i) {       // ...
            i.play();
 }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)  tune(e[i]);
  }
  public static void main(String[] args) {
     Instrument[] orchestra = new Instrument[5];
     int i = 0;
     orchestra[i++] = new Wind();
     orchestra[i++] = new Percussion();
     orchestra[i++] = new Stringed();
     orchestra[i++] = new Brass();
     orchestra[i++] = new Woodwind();
     tuneAll(orchestra); } }
```

Upcasting during addition to the array

# Επεξηγήσεις



Παρατηρείστε ότι:

- Δεν έχει σημασία αν γίνεται upcasting σε κανονική κλάση, σε αφηρημένη κλάση ή σε διαπροσωπεία.

Η δημοσιότητα της Διαπροσωπείας:

- Οι διαπροσωπείες μπορούν να προσδιοριστούν ως **δημόσιες (public)** ή **φιλικές**, ακολουθώντας τους ίδιους κανόνες που ακολουθούν οι κανονικές κλάσεις.
- Οι μέθοδοι μιας διαπροσωπείας εξυπακούονται σαν δημόσιες (αν δεν προσδιοριστούν ρητώς έτσι).
- Συνεπώς, κατά την *υλοποίηση* της διαπροσωπείας με κάποια κλάση, οι μέθοδοι της κλάσης **πρέπει** να δηλωθούν ως δημόσιες.
- Διαφορετικά προκύπτει περιορισμός της πρόσβασης σε κάποια μέθοδο κατά την κληρονομικότητα, πράγμα που απαγορεύει ο μεταφραστής της Java.

# Πλήρης αποσύνδεση υλοποίησης-

ΕΠΛ233

- "Strategy" design pattern: αφορά στη δημιουργία μεθόδου, η οποία συμπεριφέρεται με διαφορετικό τρόπο ανάλογα με τα αντικείμενα που περνιούνται στη μέθοδο αυτή σαν παράμετροι.
  - Η μέθοδος περιλαμβάνει το "σταθερό" τμήμα του αλγορίθμου που πρέπει να εκτελεσθεί.
  - Η "στρατηγική" αφορά στη διαφοροποίηση της συμπεριφοράς της μεθόδου και αντιστοιχεί σε ένα αντικείμενο που περνιέται σαν παράμετρος στην μέθοδο, και το οποίο *περιέχει κώδικα προς εκτέλεση*.
- Παράδειγμα:
  - Μέθοδος Apply.process() δέχεται σαν παραμέτρους ένα αντικείμενο "Processor" και ένα αντικείμενο "Object".
  - Ο "Processor" υλοποιεί την "στρατηγική", η οποία διαφοροποιεί την συμπεριφορά της κλάσης Apply.

# Παράδειγμα



```
import java.util.*;

  class Processor {
    public String name() {
      return getClass().getSimpleName();
    }
    Object process(Object input) { return input; }
  }

  class Upcase extends Processor {
```

```
class Splitter extends Processor {
  String process(Object input) {
    // The split() argument divides a String into pieces:
    return Arrays.toString(((String)input).split(" "));
  }
}

public class Apply {
  public static void process(Processor p, Object s) {
    print("Using Processor " + p.name());
    print(p.process(s));
  }
  public static String s =
    "Disagreement with beliefs is by definition incorrect";
  public static void main(String[] args) {
    process(new Upcase(), s);
    process(new Downcase(), s);
```

# Παράδειγμα (συνέχεια)

ΕΠΛ233

- Η παραπάνω μέθοδος (Apply.process) δεν μπορεί ωστόσο να εφαρμοστεί σε κλάσεις που *δεν* κληρονομούν από την Processor, ακόμα κι αν έχουν την ίδια διαπροσωπεία. Π.χ.:

```
public class Filter {
  public String name() {
    return getClass().getSimpleName();
  }
  public Waveform process(Waveform input) {
       return input;
   }
}
```

- Η μή δυνατότητα χρήσης της Filter σε συνδυασμό με την Apply.process ώστε να γίνει επαναχρησιμοποίηση του κώδικα της Apply.process για τις ανάγκες

# Παράδειγμα (συνέχεια)

ΕΠΛ233

```
public interface Processor {
  String name();
  Object process(Object input);
}

public class Apply {
  public static void process(Processor p, Object s) {
    print("Using Processor " + p.name());
    print(p.process(s));
  }
}
```

- Η “επαναχρησιμοποίηση” του κώδικα της Apply μπορεί να γίνει εφόσον οι προγραμματιστές γράφουν τις κλάσεις τους λαμβάνοντας υπόψη τη διαπροσωπεία Processor.

# Επαναχρησιμοποίηση Interface

ΕΠΛ233

```
public abstract class StringProcessor implements Processor{
  public String name() {
    return getClass().getSimpleName();
  }
  public abstract String process(Object input);
  public static String s =
    "If she weighs the same as a duck, she's made of wood";

public static void main(String[] args) {
    Apply.process(new Upcase(), s);
    Apply.process(new Downcase(), s);
    Apply.process(new Splitter(), s);
  }
}
```

# Επαναχρησιμοποίηση Interface

ΕΠΛ233

```
class Upcase extends StringProcessor {
  public String process(Object input) { // Covariant return
    return ((String)input).toUpperCase();
  }
}

class Downcase extends StringProcessor {
  public String process(Object input) {
    return ((String)input).toLowerCase();
  }
}

class Splitter extends StringProcessor {
  public String process(Object input) {
    return Arrays.toString(((String)input).split(" "));
  }
```

# Άλλοι τρόποι επαναχρησιμοποίησης


- Συχνά δεν υπάρχει δυνατότητα να τροποποιήσουμε κάποιες κλάσεις, τις οποίες θέλουμε να χρησιμοποιήσουμε:
  - Αν π.χ. οι κλάσεις αυτές ανήκουν σε βιβλιοθήκες τις οποίες δεν έχουμε υλοποιήσει εμείς οι ίδιοι.

- Στις περιπτώσεις αυτές μπορούμε να “προσαρμόσουμε” τις κλάσεις που μας ενδιαφέρουν στη Διαπροσωπεία που χρειαζόμαστε, χρησιμοποιώντας το Adapter design pattern.

- Στο ακόλουθο παράδειγμα, θα εφαρμόσουμε το Adapter στις κλάσεις Filter.

# Filter.java: η κλάση που μας

ΕΠΛ233

```
public class Filter {
  public String name() {
    return getClass().getSimpleName();
  }
  public Waveform process(Waveform input) { return input; }
}

public class Waveform {
  private static long counter;
  private final long id = counter++;
  public String toString() {
      return "Waveform " + id;
  }
}
```

# Filter.java

ΕΠΛ233

```
public class LowPass extends Filter {
  double cutoff;
  public LowPass(double cutoff) {
      this.cutoff = cutoff;
  }
  public Waveform process(Waveform input) {
    return input; // Dummy processing
  }
}
public class HighPass extends Filter {
  double cutoff;
  public HighPass(double cutoff) { this.cutoff = cutoff; }
  public Waveform process(Waveform input) { return input; }
}
```

# Filter.java

ΕΠΛ233

```
public class BandPass extends Filter {
  double lowCutoff, highCutoff;
  public BandPass(double lowCut, double highCut) {
    lowCutoff = lowCut;
    highCutoff = highCut;
  }
  public Waveform process(Waveform input) { return input; }
}
```

# FilterAdapter

ΕΠΛ233

```
class FilterAdapter implements Processor {
  Filter filter;
  public FilterAdapter(Filter filter) {
    this.filter = filter;
  }
  public String name() { return filter.name(); }
  public Waveform process(Object input) {
    return filter.process((Waveform)input);
  }
}

public class FilterProcessor {
  public static void main(String[] args) {
    Waveform w = new Waveform();
    Apply.process(new FilterAdapter(new LowPass(1.0)), w);
    Apply.process(new FilterAdapter(new HighPass(2.0)), w);
    Apply.process(
```

# «Πολλαπλή» Κληρονομικότητα

ΕΠΛ233

- Μια Διαπροσωπεία (interface), δεν έχει καθόλου υλοποίηση και δεν δεσμεύει μνήμη.
- Συνεπώς, μπορούμε να συνδυάσουμε πολλές διαπροσωπείες μαζί στην κληροδότηση κάποιας κλάσης.
- Στην C++, μια κλάση μπορεί να κληρονομεί από περισσότερες της μιας κλάσης.
- Αυτό δεν είναι δυνατόν στην JAVA. Μπορούμε ωστόσο να συνδυάσουμε πολλές διαπροσωπείες μαζί, τις οποίες να υλοποιεί από κοινού μια κλάση της JAVA, δίνοντάς μας μια μορφή **πολλαπλής κληρονομικότητας** (multiple

# Παράδειγμα Πολλαπλής Κληρον.

ΕΠΛ233

```
import java.util.*;

interface CanFight {void fight();}

interface CanSwim {void swim();}

interface CanFly {void fly();}

class ActionCharacter {
  public void fight() {}
}
class Hero extends ActionCharacter
    implements CanFight,CanSwim,CanFly {
  public void swim() {}
  public void fly() {}
}
```

# Παράδειγμα Πολλαπλής Κληρον.

ΕΠΛ233

```
public class Adventure {
  static void t(CanFight x) { x.fight();}
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) {          x.fly();}
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero i = new Hero();
    t(i); // Treat it as a CanFight
    u(i); // Treat it as a CanSwim
    v(i); // Treat it as a CanFly
    w(i); // Treat it as an ActionCharacter }}
```

# Η χρησιμότητα των Διαπροσωπειών


- Μας παρέχουν τη δυνατότητα να κάνουμε upcasting σε περισσότερους από έναν τύπους-βάσης, όπως στο προηγούμενο παράδειγμα.
- Αποτρέπουν τη δημιουργία αντικειμένων για κάποια συγκεκριμένη κλάση (όπως και όταν μια κλάση είναι αφηρημένη).
- Προτιμάτε τις διαπροσωπείες από τις αφηρημένες κλάσεις, στις περιπτώσεις που θέλετε να κατασκευάσετε βασικές κλάσεις κληροδότες, χωρίς οποιουσδήποτε ορισμούς μεθόδων και πεδία δεδομένων.
- Οι Διαπροσωπείες μπορούν να επεκταθούν με τη χρήση κληρονομικότητας

# Κληρονομικότητα στις Διαπροσωπείες


```
interface Monster { void menace();}
interface DangerousMonster extends Monster { void destroy();}
interface Lethal { void kill();}
class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}
interface Vampire extends DangerousMonster,Lethal{
   void drinkBlood();
}
class VeryBadVampire implements Vampire {
    public void menace() {}
    public void destroy() {}
    public void kill() {}
    public void drinkBlood() {}
}
```

# Κληρονομικότητα στις Διαπροσωπείες


```
public class HorrorShow {
   static void u(Monster b) { b.menace(); }
   static void v(DangerousMonster d) {
      d.menace();
      d.destroy();
   }
  static void w(Lethal l) { l.kill(); }

  public static void main(String[] args) {
      DangerousMonster barney = new DragonZilla();
      u(barney);
      v(barney);
      Vampire vlad = new VeryBadVampire();
      u(vlad);    v(vlad);    w(vlad);
}}
```

# Συγκρούσεις ονομάτων σε συνδυασμό διαπροσωπειών



```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; }
}
class C3 extends C implements I2 { public int f(int i) { return 1; }}
class C4 extends C implements I3 { public int f() { return 1; }  }
class C5 extends C implements I1 {}
interface I4 extends I1, I3 {}
```

# Αρχικοποιήσεις μελών Διαπροσωπείας



- Τα πεδία των διαπροσωπειών έχουν προκαθοριστεί να είναι στατικά και τελικά. Ωστόσο, μπορούμε να τα αρχικοποιούμε με μη-σταθερές εκφράσεις.
- Η αρχικοποίηση γίνεται την στιγμή που φορτώνεται η διαπροσωπεία, όταν δηλ. γίνει αναφορά στα πεδία της.

```
import java.util.*;
public interface RandVals {
  Random rand = new Random();
  int randomInt = rand.nextInt(10);
  long randomLong = rand.nextLong() * 10;
  float randomFloat = rand.nextLong() * 10;
  double randomDouble = rand.nextDouble() * 10;
}
```