# Αντικειμενοστρέφεια

Henri Matisse, *Harmony in Red,* 1908

Κωστής Σαγώνας <kostis@cs.ntua.gr>

# Ορισμοί αντικειμενοστρέφειας

- Ποιοι είναι οι ορισμοί των παρακάτω;
  - Αντικειμενοστρεφής γλώσσα προγραμματισμού
  - Αντικειμενοστρεφής προγραμματισμός
- Αλλά από την άλλη μεριά, για ποιο λόγο να τους ξέρουμε;

## Κάποιες γενικές παρατηρήσεις:

- Ο αντικειμενοστρεφής προγραμματισμός δεν είναι απλά προγραμματισμός σε μια αντικειμενοστρεφή γλώσσα
- Οι αντικειμενοστρεφείς γλώσσες δεν είναι όλες σαν τη Java

# Περιεχόμενα

- Αντικειμενοστρεφής προγραμματισμός στην ML
- Μη αντικειμενοστρεφής προγραμματισμός στη Java

Χαρακτηριστικά αντικειμενοστρεφών γλωσσών

- Κλάσεις
- Πρωτότυπα
- Κληρονομικότητα (inheritance)
- Ενθυλάκωση (encapsulation)
- Πολυμορφισμός

```
public class Node {
  private String data;
  private Node link;
  public Node(String theData, Node theLink) {
    data = theData;
    link = theLink;
  }
  public String getData() {
    return data;
  }
  public Node getLink() {
    return link;
  }
}
```

Ένα προηγούμενο παράδειγμα σε Java: ένας κόμβος που μπορεί να χρησιμοποιηθεί για την κατασκευή μιας στοίβας από συμβολοσειρές

# Η κλάση `Node`

- Δύο πεδία: **`data`** και **`link`**
- Ένας κατασκευαστής που αναθέτει αρχικές τιμές στα πεδία **`data`** και **`link`**
- Δύο μέθοδοι: **`getData`** και **`getLink`**
- Σε κάποιο αφηρημένο επίπεδο μπορούμε να θεωρήσουμε ότι ένα αντικείμενο δέχεται ένα μήνυμα (π.χ. το μήνυμα “get data” ή “get link”) και επιστρέφει μια απάντηση στο μήνυμα (π.χ. κάποιο String ή κάποιο άλλο αντικείμενο)
- Δηλαδή ένα αντικείμενο συμπεριφέρεται σα μια συνάρτηση με τύπο **`message -> response`**

# Αντικειμενοστρεφής προγραμματισμός στην ML

- Την ίδια ιδέα μπορούμε να την υλοποιήσουμε και σε ML
- Ορίζουμε τύπους για τα μηνύματα και τις απαντήσεις τους

```
datatype message = GetData | GetLink;

datatype response =
    Data of string
  | Object of message -> response;
```

- Για την κατασκευή ενός κόμβου καλούμε τη συνάρτηση **node**, περνώντας τις δύο πρώτες παραμέτρους
- Το αποτέλεσμα είναι μια συνάρτηση με τύπο **message -> response**

```
fun node data link GetData = Data data
  | node data link GetLink = Object link;
```

# Παράδειγμα χρήσης της node

```
- val n1 = node "Hello" null;
val n1 = fn : message -> response
- val n2 = node "world" n1;
val n2 = fn : message -> response
- n1 GetData;
val it = Data "Hello" : response
- n2 GetData;
val it = Data "world" : response
```

- Αντικείμενα που αποκρίνονται σε μηνύματα
- Το **`null`** πρέπει να είναι ένα αντικείμενο του τύπου **`message -> response`**, όπως π.χ.

```
fun null _ = Data "null";
```

# Η κλάση `Stack`

- Ένα πεδίο: **`top`**
- Τρεις μέθοδοι: **`hasMore`**, **`add`**, και **`remove`**
- Υλοποιείται με χρήση μιας συνδεδεμένης λίστας από αντικείμενα **`Node`**

Η επέκταση των μηνυμάτων και των απαντήσεων, τόσο για **node** όσο και για **stack**

```
datatype message =
    IsNull
  | Add of string
  | HasMore
  | Remove
  | GetData
  | GetLink;

datatype response =
    Pred of bool
  | Data of string
  | Removed of (message -> response) * string
  | Object of message -> response;

fun root _ = Pred false;
```

Η **root** χειρίζεται όλα τα μηνύματα με το να επιστρέφει **Pred false**

```
fun null IsNull = Pred true
  | null message = root message;

fun node data link GetData = Data data
  | node data link GetLink = Object link
  | node   _    _  message = root message;

fun stack top HasMore =
      let val Pred(p) = top IsNull
      in Pred(not p) end
  | stack top (Add data) =
      Object(stack (node data top))
  | stack top Remove =
      let
        val Object(next) = top GetLink
        val Data(data) = top GetData
      in
        Removed(stack next, data)
      end
  | stack _ message = root message;
```

```
- val a = stack null;
val a = fn : message -> response
- val Object(b) = a (Add "the plow.");
val b = fn : message -> response
- val Object(c) = b (Add "forgives ");
val c = fn : message -> response
- val Object(d) = c (Add "The cut worm ");
val d = fn : message -> response
- val Removed(e,s1) = d Remove;
val e = fn : message -> response
val s1 = "The cut worm " : string
- val Removed(f,s2) = e Remove;
val f = fn : message -> response
val s2 = "forgives " : string
- val Removed(_,s3) = f Remove;
val s3 = "the plow." : string
- s1^s2^s3;
val it = "The cut worm forgives the plow." : string
```

# Κληρονομικότητα (στο περίπου...)

- Ορίζουμε μια **`peekableStack`** σαν αυτή της Java

```
fun peekableStack top Peek = top GetData
  | peekableStack top message = stack top message;
```

- Το συγκεκριμένο στυλ προγραμματισμού είναι συναφές με αυτό που υποστηρίζει η γλώσσα Smalltalk
  - Έχουμε ανταλλαγή μηνυμάτων
  - Τα μηνύματα δεν έχουν στατικούς τύπους
  - Όταν μια κλάση δε χειρίζεται κάποια μηνύματα τα προωθεί για χειρισμό στην υπερκλάση της

# Κάποιες σκέψεις

- Προφανώς, αυτός δεν είναι ο καλύτερος τρόπος να χρησιμοποιήσουμε τη γλώσσα ML
  - Τα μηνύματα και οι απαντήσεις τους δεν έχουν σωστούς τύπους
  - Δε δείξαμε ότι η ML βγάζει πολλές “binding not exhaustive” προειδοποιήσεις στα προηγούμενα παραδείγματα

- Το “ηθικό δίδαγμα” όμως είναι ότι: ακόμα και στην ML ο αντικειμενοστρεφής προγραμματισμός είναι δυνατός
- Ο αντικειμενοστρεφής προγραμματισμός δεν είναι κάτι που απαιτεί την ύπαρξη μιας αντικειμενοστρεφούς γλώσσας

**Σημείωση:** Η Objective CAML είναι μια διάλεκτος της ML που εισάγει αντικειμενοστρεφή στοιχεία στην ML

# Μη αντικειμενοστρεφής προγραμματισμός στη Java

- Παρόλο που η Java υποστηρίζει καλύτερα από την ML το αντικειμενοστρεφές στυλ προγραμματισμού, η χρήση της δεν αποτελεί εγγύηση αντικειμενοστρέφειας
  - Μπορούμε να χρησιμοποιήσουμε μόνο στατικές μεθόδους
  - Μπορούμε να βάλουμε όλον τον κώδικα σε μια μόνο κλάση
  - Μπορούμε να χρησιμοποιήσουμε τις κλάσεις σαν records ή σαν C structures–με πεδία που είναι public και χωρίς μεθόδους

```
public class Node {
  public String data; // Each node has a String...
  public Node link;   // ...and a link to the next Node
}

public class Stack {
  public Node top;    // The top node in the stack
}
```

# Μη αντικειμενοστρεφής Stack

```
public class Main {
  private static void add(Stack s, String data) {
    Node n = new Node();
    n.data = data;
    n.link = s.top;
    s.top = n;
  }
  private static boolean hasMore(Stack s) {
    return (s.top != null);
  }
  private static String remove(Stack s) {
    Node n = s.top;
    s.top = n.link;
    return n.data;
  }
  …
}
```

Παρατηρήστε τις άμεσες αναφορές σε public πεδία (δεν απαιτούνται get μέθοδοι), επίσης τα δεδομένα και ο κώδικάς τους είναι εντελώς ξεχωριστά

# Πολυμορφισμός χωρίς αντικειμενοστρέφεια

- Σε προηγούμενη διάλεξη είδαμε μια διαπροσωπεία **`Worklist`** να υλοποιείται από μια **`Stack`**, **`Queue`**, κ.λπ.

- Υπάρχει ένα κοινό τρικ υποστήριξης της συγκεκριμένης δυνατότητας πολυμορφισμού σε μη αντικειμενοστρεφείς γλώσσες
- Κάθε εγγραφή (record) αρχίζει με ένα στοιχείο κάποιας απαρίθμησης, που προσδιορίζει το είδος της **`Worklist`**

# Μη αντικειμενοστρεφής `Worklist`

```
public class Worklist {
  public static final int STACK = 0;
  public static final int QUEUE = 1;
  public static final int PRIORITYQUEUE = 2;
  public int type; // one of the above Worklist types
  public Node front; // front Node in the list
  public Node rear;  // unused when type == STACK
  public int length; // unused when type == STACK
}
```

- Το πεδίο **`type`** προσδιορίζει το είδος της **`Worklist`**
- Η ερμηνεία των άλλων πεδίων εξαρτάται από το **`type`**
- Οι μέθοδοι που διαχειρίζονται τις **`Worklist`** εγγραφές έχουν κάποια διακλάδωση με βάση την τιμή του **`type`**…

# Διακλάδωση με βάση τον τύπο

```
private static void add(Worklist w, String data) {
  if (w.type == Worklist.STACK) {
    Node n = new Node();
    n.data = data;
    n.link = w.front;
    w.front = n;
  }
  else if (w.type == Worklist.QUEUE) {
    η υλοποίηση της add για ουρές
  }
  else if (w.type == Worklist.PRIORITYQUEUE) {
    η υλοποίηση της add για ουρές με προτεραιότητα
  }
}
```

Κάθε μέθοδος που χρησιμοποιεί μια **`Worklist`** πρέπει να έχει κάποια αντίστοιχη διακλάδωση τύπου

# Μειονεκτήματα

- Η επανάληψη του κώδικα που υλοποιεί τη διακλάδωση είναι βαρετή και επιρρεπής σε προγραμματιστικά λάθη
- Ανάλογα με τη γλώσσα, μπορεί να μην υπάρχει τρόπος αποφυγής της σπατάλης χώρου εάν διαφορετικά είδη εγγραφών απαιτούν διαφορετικά πεδία
- Κάποιες τυπικές προγραμματιστικές λειτουργίες, όπως για παράδειγμα η πρόσθεση κάποιου νέου είδους/τύπου αντικειμένου, δυσκολεύουν αρκετά

# Πλεονεκτήματα αντικειμενοστρέφειας

- Όταν καλούμε μια μέθοδο κάποιας διαπροσωπείας, η υλοποίηση της γλώσσας αυτόματα αποστέλλει (dispatches) την εκτέλεση στο σωστό κώδικα της μεθόδου για το συγκεκριμένο αντικείμενο
- Οι διαφορετικές υλοποιήσεις μιας διαπροσωπείας δεν είναι απαραίτητο να μοιράζονται πεδία
- Η πρόσθεση μιας κλάσης που υλοποιεί μια διαπροσωπεία είναι πολύ εύκολη διότι δεν απαιτεί την τροποποίηση κάποιου υπάρχοντα κώδικα

# Κάποιες σκέψεις

- Ο αντικειμενοστρεφής προγραμματισμός δεν είναι το ίδιο πράγμα με τον προγραμματισμό σε μια αντικειμενοστρεφή γλώσσα
  - Μπορεί να γίνει σε μια μη αντικειμενοστρεφή γλώσσα
  - Μπορεί να μη γίνει σε μια αντικειμενοστρεφή γλώσσα
- Συνήθως, οι αντικειμενοστρεφείς γλώσσες συνδυάζονται με τα αντικειμενοστρεφή στυλ προγραμματισμού
  - *Συνήθως* ένα πρόγραμμα σε ML που έχει γραφεί με χρήση αντικειμενοστρεφούς στυλ προγραμματισμού δεν είναι ότι καλύτερο υπάρχει από πλευράς σχεδιασμού
  - Επίσης *συνήθως* ένα πρόγραμμα σε Java που έχει γραφεί με χρήση του αντικειμενοστρεφούς στυλ προγραμματισμού έχει καλύτερο σχεδιασμό από ότι ένα που έχει γραφεί χωρίς να ακολουθηθεί το συγκεκριμένο στυλ

# Χαρακτηριστικά αντικειμενοστρέφειας

# Κλάσεις

- Οι περισσότερες αντικειμενοστρεφείς γλώσσες έχουν κάποιον τρόπο ορισμού κλάσεων
- Οι κλάσεις εξυπηρετούν διάφορους σκοπούς:
  - Ομαδοποιούν πεδία και μεθόδους
  - Στιγμιοτυποποιούνται: το πρόγραμμα μπορεί να δημιουργήσει όσα αντικείμενα των κλάσεων χρειάζεται για την εκτέλεσή του
  - Αποτελούν μονάδες κληρονομικότητας: μια παραγόμενη κλάση κληρονομεί από όλες τις βασικές κλάσεις της
  - Αποτελούν τύπους: τα αντικείμενα (ή οι αναφορές τους) έχουν κάποιο όνομα κλάσης ως στατικό τύπο
  - Στεγάζουν στατικά πεδία και μεθόδους και αυτή η στέγαση γίνεται ανά κλάση, όχι ανά στιγμιότυπο
  - Λειτουργούν ως χώροι ονομάτων και ελέγχουν την ορατότητα του περιεχομένου μιας κλάσης εκτός της κλάσης

# Αντικειμενοστρέφεια χωρίς κλάσεις

- Σε μια γλώσσα με κλάσεις δημιουργούμε αντικείμενα με το να φτιάχνουμε στιγμιότυπα των κλάσεων

- Σε μια γλώσσα χωρίς κλάσεις δημιουργούμε αντικείμενα
  - είτε από το μηδέν με το να ορίσουμε τα πεδία τους και να γράψουμε τον κώδικα των μεθόδων τους
  - ή με κλωνοποίηση ενός υπάρχοντος πρωτοτύπου αντικειμένου και τροποποίηση κάποιων από τα μέρη του

```
x = new Stack();
```

Με κλάσεις:
δημιουργία
στιγμιότυπου

```
x = {
  private Node top = null;
  public boolean hasMore() {
    return (top != null);
  }
  public String remove() {
    Node n = top;
    top = n.getLink();
    return n.getData();
  }
  …
}
```

Χωρίς κλάσεις:
δημιουργία
αντικειμένου
από το μηδέν

```
x = y.clone();
x.top = null;
```

Χωρίς κλάσεις:
κλωνοποίηση
πρωτοτύπου

# Πρωτότυπα

- Ένα πρωτότυπο (prototype) είναι ένα αντικείμενο το οποίο μπορεί να αντιγραφεί ώστε να δημιουργήσει παρόμοια αντικείμενα
- Κατά τη δημιουργία αντιγράφων, το πρόγραμμα μπορεί να τροποποιήσει τις τιμές κάποιων πεδίων, να προσθέσει ή να αφαιρέσει πεδία και μεθόδους
- Οι γλώσσες που βασίζονται σε πρωτότυπα (όπως η Self) χρησιμοποιούν αυτήν την ιδέα αντί για κλάσεις

# Χωρίς κλάσεις αλλά με πρωτότυπα

- Η εύκολη δημιουργία στιγμιότυπων είναι μία από τις βασικές χρησιμότητες των κλάσεων
- Κάποια άλλα χαρακτηριστικά που πρέπει να αποχωριστούν οι γλώσσες που βασίζονται σε πρωτότυπα:
  - Τις κλάσεις ως τύπους: οι περισσότερες γλώσσες που βασίζονται σε πρωτότυπα έχουν δυναμικούς τύπους
  - Την κληρονομικότητα: οι γλώσσες που βασίζονται σε πρωτότυπα χρησιμοποιούν μια δυναμική τεχνική η οποία είναι σχετική με την κληρονομικότητα και λέγεται αντιπροσωπεία (delegation)

# Κληρονομικότητα

- Σε επιφανειακό επίπεδο, η έννοια της κληρονομικότητας είναι αρκετά εύκολη να κατανοηθεί
  - Η επέκταση κλάσεων δημιουργεί μια σχέση μεταξύ δύο κλάσεων: μια σχέση μεταξύ μιας παραγόμενης και μιας βασικής κλάσης
  - Η παραγόμενη κλάση κληρονομεί πεδία και μεθόδους από τη βασική κλάση
- Αλλά το τι κληρονομείται από τη βασική κλάση (ή κλάσεις) καθορίζεται από τη γλώσσα
- Θα δούμε ότι διαφορετικές γλώσσες έχουν διαφορετικές προσεγγίσεις πάνω σε θέματα κληρονομικότητας

# Ερωτήσεις κληρονομικότητας

- Επιτρέπονται περισσότερες από μία βασικές κλάσεις;
  - Απλή κληρονομικότητα: Smalltalk, Java
  - Πολλαπλή κληρονομικότητα : C++, CLOS, Eiffel
- Οι υποκλάσεις κληρονομούν τα πάντα;
  - Στη Java: η παραγόμενη κλάση κληρονομεί όλες τις μεθόδους και τα πεδία
  - Στη Sather: η παραγόμενη κλάση μπορεί να μετονομάσει τις κληρονομημένες μεθόδους της (κάτι που είναι χρήσιμο όταν υπάρχει πολλαπλή κληρονομικότητα), ή απλώς να τις ξε-ορίσει
- Υπάρχει κάποια καθολική βασική κλάση;
  - Μια κλάση από την οποία όλες οι κλάσεις κληρονομούν: η κλάση `Object` της Java
  - Δεν υπάρχει κάποια τέτοια κλάση στη C++

# Ερωτήσεις κληρονομικότητας

- Κληρονομούνται οι προδιαγραφές;
  - Ως υποχρεώσεις των μεθόδων για υλοποίηση, όπως στη Java
  - Ως επιπλέον προδιαγραφές: invariants, όπως στην Eiffel
- Κληρονομούνται οι τύποι;
  - Στη Java κληρονομούνται όλοι οι τύποι της βασικής κλάσης
- Υποστηρίζεται υπερκάλυψη, απόκρυψη, κ.λπ.;
  - Τι συμβαίνει στη Java, στο περίπου (χωρίς πολλές λεπτομέρειες):
    - Οι κατασκευαστές μπορούν να προσπελάσουν τους κατασκευαστές των βασικών τους κλάσεων (Αυτό γίνεται με χρήση του **`super`** που καλεί τον κατασκευαστή της άμεσης υπερκλάσης.)
    - Μια νέα μέθοδος με το ίδιο όνομα και τύπο με την κληρονομημένη μέθοδο την υπερκαλύπτει (Η υπερκαλυμμένη μέθοδος μπορεί να κληθεί με χρήση του προσδιοριστή **`super`**.)
    - Ένα νέο πεδίο ή στατική μέθοδος αποκρύπτει τις κληρονομημένες, οι οποίες όμως μπορούν να προσπελαστούν με χρήση του **`super`** ή με στατικούς τύπους της βασικής κλάσης

# Ενθυλάκωση (encapsulation)

- Απαντάται σχεδόν σε όλες τις μοντέρνες γλώσσες προγραμματισμού, όχι μόνο στις αντικειμενοστρεφείς
- Τα μέρη του προγράμματος που ενθυλακώνονται:
  - Παρουσιάζουν μια ελεγχόμενη διαπροσωπεία στους χρήστες τους
  - Αποκρύπτουν όλα τα άλλα (ιδιωτικά) μέρη τους
- Στις αντικειμενοστρεφείς γλώσσες, τα αντικείμενα ενθυλακώνονται
- (Διαφορετικές γλώσσες υλοποιούν την ενθυλάκωση με διαφορετικούς τρόπους)

# Ορατότητα των πεδίων και των μεθόδων

- Στη Java υπάρχουν τέσσερα επίπεδα ορατότητας
  - **`private`**: ορατά μόνο μέσα στην κλάση
  - (default): ορατά μόνο εντός του πακέτου (package)
  - **`protected`**: ορατά μόνο στο ίδιο πακέτο και στις παραγόμενες κλάσεις
  - **`public`**: παντού
- Κάποιες αντικειμενοστρεφείς γλώσσες (Smalltalk, Self) έχουν λιγότερα επίπεδα ελέγχου ορατότητας: όλα κοινά (**`public`**)
- Άλλες έχουν περισσότερα: στην Eiffel, πεδία και μέθοδοι μπορεί να γίνουν ορατά μόνο σε ένα συγκεκριμένο σύνολο από κλάσεις-πελάτες

# Πολυμορφισμός

- Συναντιέται σε πολλές γλώσσες, όχι μόνο στις αντικειμενοστρεφείς
- Κάποιες από τις βασικότερες εκφράσεις του πολυμορφισμού στις αντικειμενοστρεφείς γλώσσες:
  - Όταν διαφορετικές κλάσεις έχουν μεθόδους του ίδιου ονόματος και τύπου
    - Π.χ. μια κλάση στοίβας και μια κλάση ουράς μπορούν και οι δύο να έχουν μια μέθοδο ονόματι **`add`**
  - Όταν η γλώσσα επιτρέπει μια κλήση μεθόδου σε στιγμές που η κλάση του αντικειμένου δεν είναι γνωστή στατικά

# Παράδειγμα σε Java

```
public static void flashoff(Drawable d, int k) {
  for (int i = 0; i < k; i++) {
    d.show(0,0);
    d.hide();
  }
}
```

- Εδώ, η **`Drawable`** είναι μια διαπροσωπεία
- Η κλάση του αντικειμένου στην οποία αναφέρεται η αναφορά **`d`** δεν είναι γνωστή στο χρόνο μεταγλώττισης

# Δυναμική αποστολή (dynamic dispatch)

- Στη Java, ο στατικός τύπος μιας αναφοράς μπορεί να είναι μια υπερκλάση ή μια διαπροσωπεία κάποιας κλάσης
- Στο χρόνο εκτέλεσης, το σύστημα υλοποίησης της γλώσσας θα πρέπει κάπως να βρει και να καλέσει τη σωστή μέθοδο της πραγματικής κλάσης
- Αυτό αναφέρεται ως **δυναμική αποστολή (dynamic dispatch)**: η κρυφή και έμμεση διακλάδωση πάνω στην κλάση κατά την κλήση των μεθόδων
  - Η δυναμική αποστολή είναι προαιρετική στη C++
  - Χρησιμοποιείται πάντα στη Java και στις περισσότερες άλλες αντικειμενοστρεφείς γλώσσες

# Υλοποιήσεις και τύποι

- Στη Java υπάρχουν δύο μηχανισμοί:
  - Μια κλάση κληρονομεί τόσο τους τύπους όσο και την υλοποίηση της βασικής της κλάσης
  - Μια κλάση παίρνει πρόσθετους τύπους (αλλά όχι αυτόματα κάποια υλοποίηση) με την υλοποίηση διαπροσωπειών
- Το παραπάνω μερικώς διαχωρίζει την κληρονομικότητα της υλοποίησης από την κληρονομικότητα του τύπου
- Άλλες αντικειμενοστρεφείς γλώσσες διαφέρουν στο κατά πόσο ξεχωρίζουν τα δύο παραπάνω

# Υλοποιήσεις και τύποι

- Στη C++ δεν υπάρχει κάποιος αντίστοιχος διαχωρισμός
  - Υπάρχει ένας μόνο μηχανισμός για την κληρονομικότητα
  - Για την κληρονομιά τύπου μόνο, μπορούμε να χρησιμοποιήσουμε μια αφηρημένη βασική κλάση χωρίς κάποια υλοποίηση
- Από την άλλη μεριά στη Sather ο διαχωρισμός υλοποιήσεων και τύπων είναι πλήρης:
  - Μια κλάση μπορεί να δηλώσει ότι **περιλαμβάνει** κάποια άλλη κλάση, οπότε κληρονομεί την υλοποίηση αλλά όχι τον τύπο
  - Μια κλάση μπορεί να δηλώσει ότι είναι μια υποκλάση μιας αφηρημένης κλάσης, οπότε κληρονομεί τον τύπο αλλά όχι την υλοποίηση (όπως συμβαίνει με τις διαπροσωπείες στη Java)

# Χρήση δυναμικού συστήματος τύπων

- Κάποιες αντικειμενοστρεφείς γλώσσες χρησιμοποιούν δυναμικό σύστημα τύπων: π.χ. η Smalltalk και η Self
- Ένα αντικείμενο μπορεί να είναι ή να μην είναι σε θέση να απαντήσει σε ένα συγκεκριμένο μήνυμα–και αυτό μπορεί να μην μπορεί να ελεγχθεί κατά το χρόνο μετάφρασης (όπως π.χ. έγινε στο ML τρικ μας)
- Αυτό δίνει απόλυτη ελευθερία: το πρόγραμμα μπορεί να δοκιμάσει τη χρησιμοποίηση οποιασδήποτε μεθόδου σε οποιοδήποτε αντικείμενο
- (Ο πολυμορφισμός δεν έχει σχέση με τη συγκεκριμένη ελευθερία)

# Συμπερασματικά

- Σήμερα, ακολουθήσαμε μια κοσμοπολίτικη προσέγγιση:
  - Ο αντικειμενοστρεφής προγραμματισμός δεν είναι το ίδιο πράγμα με τον προγραμματισμό με μια αντικειμενοστρεφή γλώσσα
  - Οι αντικειμενοστρεφείς γλώσσες δεν είναι όλες σαν τη Java
- Το τι είναι αντικειμενοστρεφές στυλ προγραμματισμού δεν είναι μονοσήμαντα ορισμένο: υπάρχει αρκετή διαφωνία ως προς το ακριβές σύνολο των χαρακτηριστικών της αντικειμενοστρέφειας και τα χαρακτηριστικά αυτά συνεχώς εξελίσσονται
- Δείξτε σκεπτικισμό στους ορισμούς!