

# Περισσότερη Java



Franz Marc, *Fate of the animals*, 1913

Κωστής Σαγώνας <[kostis@cs.ntua.gr](mailto:kostis@cs.ntua.gr)>  
Νίκος Παπασπύρου <[nickie@softlab.ntua.gr](mailto:nickie@softlab.ntua.gr)>

# Πολυμορφισμός υποτύπων

---

**Person p;**

- Είναι το παραπάνω μια δήλωση ότι το `p` είναι μια αναφορά σε ένα αντικείμενο της κλάσης `Person`;
- Όχι ακριβώς – ο *τύπος* `Person` μπορεί να περιλαμβάνει αναφορές σε αντικείμενα άλλων κλάσεων
- Αυτό διότι η Java υποστηρίζει **πολυμορφισμό υποτύπων (subtype polymorphism)**

# Περιεχόμενα

---

- Υλοποίηση διαπροσωπείας των κλάσεων
- Επέκταση των κλάσεων
- Επέκταση και υλοποίηση
- Πολλαπλή κληρονομικότητα
- Παραμετρικότητα μέσω γενικών μεθόδων (generics)

# Διαπροσωπείες (interfaces)

---

- Ένα **πρωτότυπο μεθόδου (method prototype)** απλώς δίνει το όνομα της μεθόδου και τον τύπο της – όχι το σώμα της
- Οι διαπροσωπείες είναι συλλογές από πρωτότυπα μεθόδων

```
public interface Drawable {  
    void show(int xPos, int yPos);  
    void hide();  
}
```

- Μια κλάση μπορεί να δηλώσει ότι υλοποιεί μια συγκεκριμένη διαπροσωπεία
- Μετά πρέπει να παρέχει ορισμούς **public** μεθόδων οι οποίοι ταιριάζουν με εκείνους της διαπροσωπείας

# Παραδείγματα

---

```
public class Icon implements Drawable {  
    public void show(int x, int y) {  
        ... σώμα της μεθόδου ...  
    }  
    public void hide() {  
        ... σώμα της μεθόδου ...  
    }  
    ...περισσότερες μέθοδοι και πεδία...  
}
```

```
public class Square implements Drawable, Scalable {  
    ... πρέπει να υλοποιεί όλες τις μεθόδους όλων των διαπρωπειών ...  
}
```

# Γιατί χρησιμοποιούμε διαπρωπείες;

---

- Μια διαπρωπεία μπορεί να υλοποιείται από πολλές διαφορετικές κλάσεις:

```
public class Window implements Drawable ...  
public class Icon implements Drawable ...  
public class Oval implements Drawable ...
```

- Το όνομα της διαπρωπείας μπορεί να χρησιμοποιηθεί ως ένας τύπος αναφοράς:

```
Drawable d;  
d = new Icon("i1.gif");  
d.show(0,0);  
d = new Oval(20,30);  
d.show(0,0);
```

# Πολυμορφισμός με διαπρωπείες

---

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

- Η παραπάνω μέθοδος είναι πολυμορφική: η κλάση του αντικειμένου που αναφέρεται από την παράμετρο `d` δεν είναι γνωστή κατά το χρόνο μετάφρασης
- Το μόνο που είναι γνωστό είναι ότι είναι μια κλάση που υλοποιεί τη διαπρωπεία `Drawable` (**implements Drawable**) και κατά συνέπεια είναι μια κλάση που έχει μεθόδους `show` και `hide` οι οποίες μπορούν να κληθούν

## Ένα πιο ολοκληρωμένο παράδειγμα

---

- Η επόμενη διαφάνεια δείχνει τη διαπροσωπεία μιας κλάσης `Worklist` που είναι μια συλλογή από αντικείμενα `String`
- Η κλάση περιέχει μεθόδους με τις οποίες μπορούμε
  - να προσθέσουμε ένα αντικείμενο στη συλλογή,
  - να αφαιρέσουμε ένα αντικείμενο από τη συλλογή, και
  - να ελέγξουμε κατά πόσο μια συλλογή είναι κενή ή όχι

```
public interface Worklist {
    /**
     * Add one String to the worklist.
     * @param item the String to add
     */
    void add(String item);

    /**
     * Test whether there are more elements in the
     * worklist: that is, test whether more elements
     * have been added than have been removed.
     * @return true iff there are more elements
     */
    boolean hasMore();

    /**
     * Remove one String from the worklist and return it.
     * There must be at least one element in the worklist.
     * @return the String item removed
     */
    String remove();
}
```

## Σχόλια στις διαπροσωπείες

---

- Η ύπαρξη σχολίων είναι σημαντική για μια διαπροσωπεία, διότι δεν υπάρχει κώδικας ώστε ο αναγνώστης/χρήστης να καταλάβει τι (σκοπεύει να) κάνει η κάθε μέθοδος
- Η διαπροσωπεία της `Worklist` δεν προσδιορίζει κάποια συγκεκριμένη δομή ή ταξινόμηση: μπορεί να υλοποιείται από μια στοίβα, μια ουρά, ή κάτι άλλο
- Θα την υλοποιήσουμε ως στοίβα, μέσω συνδεδεμένης λίστας

```
/**
 * A Node is an object that holds a String and a link
 * to the next Node. It can be used to build linked
 * lists of Strings.
 */
public class Node {
    private String data; // Each node has a String...
    private Node link; // and a link to the next Node

    /**
     * Node constructor.
     * @param theData the String to store in this Node
     * @param theLink a link to the next Node
     */
    public Node(String theData, Node theLink) {
        data = theData;
        link = theLink;
    }
}
```

```
/**
 * Accessor for the String data stored in this Node.
 * @return our String item
 */
public String getData() {
    return data;
}

/**
 * Accessor for the link to the next Node.
 * @return the next Node
 */
public Node getLink() {
    return link;
}
}
```

```
/**
 * A Stack is an object that holds a collection of
 * Strings.
 */
public class Stack implements Worklist {
    private Node top = null; // top Node in the stack

    /**
     * Push a String on top of this stack.
     * @param data the String to add
     */
    public void add(String data) {
        top = new Node(data, top);
    }
}
```

```
/**
 * Test whether this stack has more elements.
 * @return true if this stack is not empty
 */
public boolean hasMore() {
    return (top != null);
}

/**
 * Pop the top String from this stack and return it.
 * This should be called only if the stack is
 * not empty.
 * @return the popped String
 */
public String remove() {
    Node n = top;
    top = n.getLink();
    return n.getData();
}
}
```

## Ένα παράδειγμα χρήσης

---

```
Worklist w;  
w = new Stack();  
w.add("ο Παρασκευάς.");  
w.add("βας, ");  
w.add("Βας, βας,");  
System.out.print(w.remove());  
System.out.print(w.remove());  
System.out.println(w.remove());
```

- Έξοδος: Βας, βας, βας, ο Παρασκευάς.
- Άλλες υλοποιήσεις της κλάσης `Worklist` είναι πιθανές: με χρήση `Queue`, `PriorityQueue`, κ.α.

# Επέκταση των κλάσεων

## Περισσότερος πολυμορφισμός

---

- Θα δούμε μια άλλη, πιο πολύπλοκη, πηγή πολυμορφισμού
- Μια κλάση μπορεί να παράγεται από μια άλλη, με χρήση της λέξης κλειδί **extends**

Για παράδειγμα: θα ορίσουμε μια κλάση **PeekableStack** η οποία είναι σαν την κλάση **Stack**, αλλά έχει επίσης μια μέθοδο **peek** που εξετάζει το στοιχείο στην κορυφή της στοίβας χωρίς όμως να το αφαιρεί από αυτή

```
/**
 * A PeekableStack is an object that does everything
 * a Stack can do, and can also peek at the top
 * element of the stack without popping it off.
 */
public class PeekableStack extends Stack {

    /**
     * Examine the top element on the stack, without
     * popping it off. This should be called only if
     * the stack is not empty.
     * @return the top String from the stack
     */
    public String peek() {
        String s = remove();
        add(s);
        return s;
    }
}
```

## Κληρονομικότητα (inheritance)

---

- Επειδή η κλάση `PeekableStack` επεκτείνει την κλάση `Stack`, κληρονομεί όλες τις μεθόδους και τα πεδία της (Κάτι τέτοιο δε συμβαίνει με τις διαπροσωπείες: όταν μια κλάση υλοποιεί μια διαπροσωπεία, το μόνο που αναλαμβάνει είναι μια υποχρέωση να υλοποιήσει κάποιες μεθόδους.)
- Εκτός από κληρονομικότητα, η επέκταση των κλάσεων οδηγεί και σε πολυμορφισμό

```
Stack s1 = new PeekableStack();  
PeekableStack s2 = new PeekableStack();  
s1.add("drive");  
s2.add("cart");  
System.out.println(s2.peek());
```

Προσέξτε ότι μια κλήση `s1.peek()` δε θα ήταν νόμιμη, παρόλο που η `s1` είναι μια αναφορά σε ένα αντικείμενο της κλάσης `PeekableStack`. Οι λειτουργίες που επιτρέπονται στη Java καθορίζονται από το στατικό τύπο της αναφοράς και όχι από την κλάση του αντικειμένου.

# Ερώτηση

---

- Η υλοποίηση της μεθόδου `peek` δεν ήταν η πιο αποδοτική:

```
public String peek() {  
    String s = remove();  
    add(s);  
    return s;  
}
```

- Γιατί δεν κάνουμε το παρακάτω;

```
public String peek() {  
    return top.getData();  
}
```

## Απάντηση

---

- Το πεδίο `top` της κλάσης `Stack` έχει δηλωθεί `private`
- Η κλάση `PeekableStack` δε μπορεί να το προσπελάσει
- Για μια πιο αποδοτική μέθοδο `peek`, η κλάση `Stack` πρέπει να καταστήσει το πεδίο `top` ορατό στις κλάσεις που την επεκτείνουν
- Δηλαδή πρέπει να το δηλώσει ως `protected` αντί για `private`
- Συνήθης πρόκληση σχεδιασμού για αντικειμενοστρεφείς γλώσσες: πως ο σχεδιασμός θα κάνει εύκολη την επαναχρησιμοποίηση μεθόδων μέσω κληρονομικότητας

# Αλυσίδες κληρονομικότητας

---

- Στη Java, μια κλάση μπορεί να έχει πολλές κλάσεις που παράγονται από αυτή
- Για την ακρίβεια, όλες οι κλάσεις της Java (εκτός από μία) παράγονται από κάποια άλλη κλάση
- Εάν στον ορισμό μιας κλάσης δεν προσδιορίζεται κάποια πρόταση **extends**, η Java αυτόματα εννοεί:  
**extends Object**
- Η κλάση **Object** είναι η πρωταρχική κλάση της Java (δεν παράγεται από κάποια άλλη)

# Η κλάση `Object`

---

- Όλες οι κλάσεις παράγονται, άμεσα ή έμμεσα, από την προκαθορισμένη κλάση `Object`
  - (εκτός φυσικά από την κλάση `Object`)
- Όλες οι κλάσεις κληρονομούν μεθόδους από την κλάση `Object`, για παράδειγμα:
  - `getClass`, επιστρέφει την κλάση του αντικειμένου
  - `toString`, για μετατροπή του αντικειμένου σε `String`
  - `equals`, για σύγκριση με άλλα αντικείμενα
  - `hashCode`, για υπολογισμό ενός ακεραίου (`int`) που αντιστοιχεί στην τιμή του κωδικού κατακερματισμού (hash code) του αντικειμένου
  - κ.λπ.

## Υπερκάλυψη κληρονομημένων ορισμών

---

- Κάποιες φορές μπορεί να θέλουμε να επανακαθορίσουμε τη λειτουργικότητα μια κληρονομημένης μεθόδου
- Αυτό δε γίνεται με χρήση κάποιου ειδικού κατασκευαστή: ένας νέος ορισμός μιας μεθόδου αυτόματα **υπερκαλύπτει (overrides)** έναν κληρονομημένο ορισμό του ίδιου ονόματος και τύπου

## Παράδειγμα υπερκάλυψης

---

- Η κληρονομημένη μέθοδος `toString` απλώς συνδυάζει το όνομα της κλάσης και τον κωδικό κατακερματισμού (σε μορφή δεκαεξαδικού αριθμού)
- Με άλλα λόγια, ο κώδικας της default μεθόδου τυπώνει κάτι σαν το εξής: `Stack@b3d`
- Μια ειδική μέθοδος `toString` στη μέθοδο `Stack`, σαν την παρακάτω, μπορεί να τυπώσει ένα πιο διευκρινιστικό μήνυμα:

```
public String toString() {  
    return "Stack with top at " + top;  
}
```

# Ιεραρχίες κληρονομικότητας

---

- Η σχέση κληρονομικότητας δημιουργεί μια ιεραρχία
- Η ιεραρχία αυτή είναι ένα δένδρο με ρίζα την κλάση **Object**
- Σε κάποιες περιπτώσεις οι κλάσεις απλώς επεκτείνουν η μία την άλλη
- Σε άλλες περιπτώσεις, η ιεραρχία των κλάσεων και η κληρονομικότητα χρησιμοποιούνται ούτως ώστε ο κώδικας που είναι κοινός για περισσότερες από μία κλάσεις να υπάρχει μόνο σε μια κοινή βασική κλάση

Δύο κλάσεις με πολλά κοινά στοιχεία — αλλά καμία δεν είναι μια απλή επέκταση της άλλης.

```
public class Label {
    private int x, y;
    private int width;
    private int height;
    private String text;
    public void move
        (int newX, int newY)
    {
        x = newX;
        y = newY;
    }
    public String getText()
    {
        return text;
    }
}
```

```
public class Icon {
    private int x, y;
    private int width;
    private int height;
    private Gif image;
    public void move
        (int newX, int newY)
    {
        x = newX;
        y = newY;
    }
    public Gif getImage()
    {
        return image;
    }
}
```

Ο κώδικας και τα δεδομένα που είναι κοινά έχουν εξαχθεί σε μια κοινή “βασική” κλάση.

```
public class Graphic {
    protected int x,y;
    protected int width,height;
    public void move(int newX, int newY) {
        x = newX;
        y = newY;
    }
}
```

```
public class Label
    extends Graphic {
    private String text;
    public String getText()
    {
        return text;
    }
}
```

```
public class Icon
    extends Graphic {
    private Gif image;
    public Gif getImage()
    {
        return image;
    }
}
```

## Ένα πρόβλημα σχεδιασμού

---

- Πολλές φορές όταν γράφουμε τον ίδιο κώδικα ξανά και ξανά, σκεφτόμαστε ότι ο κώδικας αυτός πρέπει να “βγει” σε μια συνάρτηση (σε μία μέθοδο)
- Όταν γράψουμε τις ίδιες μεθόδους ξανά και ξανά, σκεφτόμαστε ότι κάποια μέθοδος πρέπει να “βγει” σε μια κοινή βασική κλάση
- Οπότε είναι καλό να καταλάβουμε νωρίς στο σχεδιασμό κατά πόσο υπάρχει ανάγκη για κοινές βασικές κλάσεις, πριν γράψουμε αρκετό κώδικα ο οποίος χρειάζεται αναδιοργάνωση

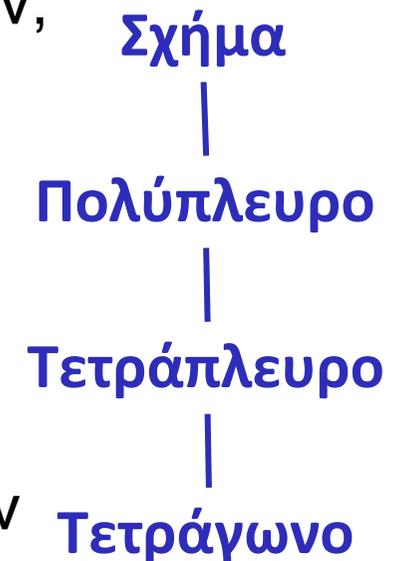
# Υποτύποι και κληρονομικότητα

---

- Μια παραγόμενη κλάση είναι ένας υποτύπος
- Από προηγούμενη διάλεξη:

*Ένας υποτύπος είναι ένα υποσύνολο των τιμών κάποιου τύπου, αλλά υποστηρίζει ένα υπερσύνολο των λειτουργιών του.*

- Κατά το σχεδιασμό της ιεραρχίας των κλάσεων, πρέπει να σκεφτόμαστε την κληρονομικότητα της λειτουργικότητάς τους
- Όμως οι “φυσικές” ιεραρχίες δεν είναι πάντα ό,τι πιο κατάλληλο μπορεί να υπάρξει όσον αφορά στην κληρονομικότητα των λειτουργιών



# Επέκταση και υλοποίηση

# Επέκταση και υλοποίηση

---

- Οι κλάσεις μπορούν να χρησιμοποιήσουν τις λέξεις κλειδιά **extends** και **implements** συγχρόνως
- Για κάθε κλάση, η υλοποίηση ενός συστήματος Java κρατάει πληροφορίες για αρκετές ιδιότητες, όπως για παράδειγμα:

A: τις διαπροσωπείες που η κλάση υλοποιεί

B: τις μεθόδους που είναι υποχρεωμένη να ορίσει

Γ: τις μεθόδους που ορίζονται για την κλάση

Δ: τα πεδία που περιλαμβάνει η κλάση

## Απλές περιπτώσεις

---

- Ένας ορισμός μεθόδου επηρεάζει μόνο το Γ
- Ένας ορισμός πεδίου επηρεάζει μόνο το Δ
- Μια δήλωση **implements** επηρεάζει τα A και B
  - Όλες οι διαπροσωπείες προσθέτονται στο A
  - Όλες οι μέθοδοι τους προσθέτονται στο B

A: τις διαπροσωπείες που η κλάση υλοποιεί

B: τις μεθόδους που είναι υποχρεωμένη να ορίσει

Γ: τις μεθόδους που ορίζονται για την κλάση

Δ: τα πεδία που περιλαμβάνει η κλάση

## Η δύσκολη περίπτωση

---

- Μια δήλωση **extends** επηρεάζει όλες τις πληροφορίες:
  - Όλες οι διαπροσωπείες της βασικής κλάσης προσθέτονται στο A
  - Όλες οι μέθοδοι που υποχρεούται η βασική κλάση να ορίσει προσθέτονται στο B
  - Όλες οι μέθοδοι της βασικής κλάσης προσθέτονται στο Γ
  - Όλα τα πεδία της βασικής κλάσης προσθέτονται στο Δ

A: τις διαπροσωπείες που η κλάση υλοποιεί

B: τις μεθόδους που είναι υποχρεωμένη να ορίσει

Γ: τις μεθόδους που ορίζονται για την κλάση

Δ: τα πεδία που περιλαμβάνει η κλάση

# Το προηγούμενο παράδειγμά μας

---

```
public class Stack implements Worklist {...}
public class PeekableStack extends Stack {...}
```

- Η κλάση `PeekableStack` έχει ως:
  - Α: τη διαπροσωπεία `Worklist`, από κληρονομιά
  - Β: τις υποχρεώσεις για υλοποίηση των μεθόδων `add`, `hasMore`, και `remove`, επίσης από κληρονομιά
  - Γ: τις μεθόδους `add`, `hasMore`, και `remove`, κληρονομημένες, όπως επίσης και τη δική της μέθοδο `peek`
  - Δ: το πεδίο `top`, επίσης κληρονομημένο

## Μια ματιά στις `abstract` κλάσεις

---

- Παρατηρείστε ότι το  $\Gamma$  είναι υπερσύνολο του  $B$ : η κλάση πρέπει να έχει ορισμούς για όλες τις μεθόδους
- Η Java συνήθως απαιτεί το παραπάνω
- Οι κλάσεις μπορούν να απαλλαγούν από αυτήν την υποχρέωση με το να δηλωθούν αφηρημένες (**`abstract`**)
- Μια **`abstract`** κλάση μπορεί να χρησιμοποιηθεί μόνο ως βασική κλάση
- (Αυτό σημαίνει ότι δε μπορούν να δημιουργηθούν αντικείμενα της συγκεκριμένης κλάσης.)

# Τελικές (`final`) κλάσεις και μέθοδοι

---

- Περιορίζουν την κληρονομικότητα
  - Οι τελικές κλάσεις δε μπορούν να επεκταθούν και οι τελικές μέθοδοί τους δε μπορούν να ξαναοριστούν
- Παράδειγμα, η κλάση  
`java.lang.String`
- Η ύπαρξη τελικών κλάσεων είναι σημαντική για ασφάλεια
  - Ο προγραμματιστής μπορεί να ελέγξει πλήρως τη συμπεριφορά όλων των υποκλάσεων (και κατά συνέπεια των υποτύπων)

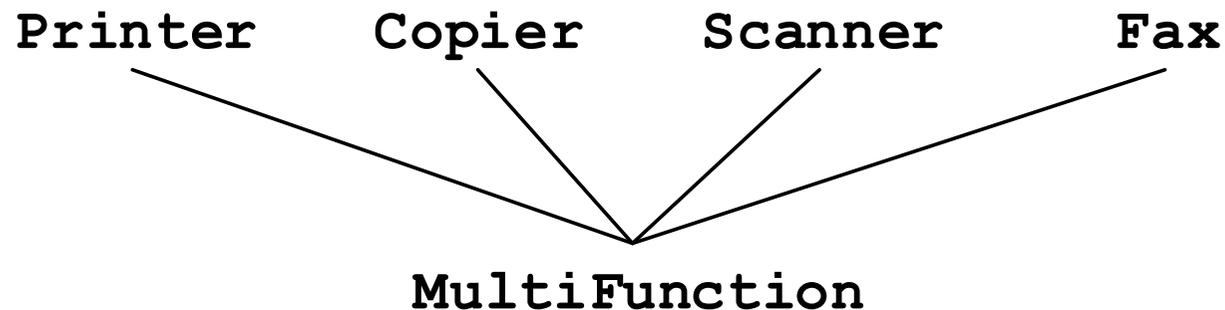
**Σημ.:** Η δήλωση `final` μπορεί να χρησιμοποιηθεί και σε πεδία: εκεί, το `final` σημαίνει ότι μπορεί να ανατεθεί τιμή μόνο μια φορά στο πεδίο

# Πολλαπλή κληρονομικότητα

## Πολλαπλή κληρονομικότητα (multiple inheritance)

---

- Σε κάποιες γλώσσες (όπως η C++) μια κλάση μπορεί να έχει περισσότερες από μία βασικές κλάσεις
- Παράδειγμα: ένα πολυμηχάνημα (multifunction printer)



- Επιφανειακά, τόσο η σημασιολογία όσο και η υλοποίηση φαίνονται εύκολες: η κλάση απλά κληρονομεί όλα τα πεδία και τις μεθόδους των βασικών της κλάσεων

## Προβλήματα συγκρούσεων

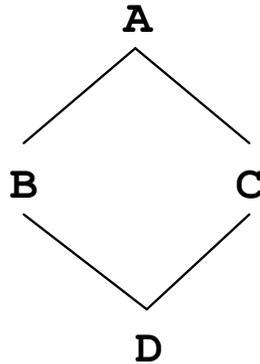
---

- Οι διαφορετικές βασικές κλάσεις είναι άσχετες μεταξύ τους και μπορεί να μην έχουν σχεδιαστεί έτσι ώστε να συνδυάζονται
- Για παράδειγμα, τόσο η κλάση **Scanner** όσο και η **Fax** μπορεί να έχουν ορίσει μια μέθοδο **transmit**
- Το ερώτημα είναι: τι πρέπει να συμβεί όταν καλέσουμε τη μέθοδο **MultiFunction.transmit**;

# Το πρόβλημα του διαμαντιού

---

- Μια κλάση μπορεί να κληρονομεί από την ίδια βασική κλάση μέσω περισσοτέρων του ενός μονοπατιού



- Εάν η κλάση **A** ορίζει ένα πεδίο **x**, τότε τόσο η **B** όσο και η **C** έχουν ένα
- Δηλαδή η κλάση **D** έχει δύο τέτοια πεδία;

## Το πρόβλημα είναι επιλύσιμο, αλλά...

---

- Μια γλώσσα που υποστηρίζει πολλαπλή κληρονομικότητα πρέπει να έχει κάποιους μηχανισμούς χειρισμού αυτών των προβλημάτων
- Βεβαίως, δεν είναι όλα τα προβλήματα τόσο πολύπλοκα
- Όμως, το βασικό ερώτημα είναι: τα πλεονεκτήματα που προσφέρει η πολλαπλή κληρονομικότητα αξίζουν την πρόσθετη πολυπλοκότητα στο σχεδιασμό της γλώσσας;
- Οι σχεδιαστές της Java ήταν (και είναι) της γνώμης ότι η πολλαπλή κληρονομικότητα δεν αξίζει τον κόπο

# Ζωή χωρίς πολλαπλή κληρονομικότητα

---

- Ένα πλεονέκτημα της πολλαπλής κληρονομικότητας είναι ότι μια κλάση μπορεί να έχει αρκετούς διαφορετικούς μεταξύ τους τύπους (π.χ. **Copier** και **Fax**)
  - Αυτό μπορεί να γίνει στη Java με χρήση διαπροσωπειών: μια κλάση μπορεί να υλοποιεί έναν απεριόριστο αριθμό από διαπροσωπείες
- Ένα επιπλέον πλεονέκτημα είναι η δυνατότητα κληρονομιάς λειτουργικότητας από πολλαπλές βασικές κλάσεις
  - Αυτό είναι δυσκολότερο να γίνει σε μια γλώσσα σαν τη Java

## Πρώθηση (forwarding)

---

```
public class MultiFunction {
    private Printer myPrinter;
    private Copier  myCopier;
    private Scanner myScanner;
    private Fax     myFax;

    public void copy() {
        myCopier.copy();
    }
    public void transmitScanned() {
        myScanner.transmit();
    }
    public void sendFax() {
        myFax.transmit();
    }
    ...
}
```

# Παραμετρικότητα μέσω Generics

## Ανυπαρξία γενικών κλάσεων στις Java 1.0-1.4

---

- Το προηγούμενο παράδειγμα κλάσης `Stack` ορίστηκε ως μια στοίβα από συμβολοσειρές
- Κατά συνέπεια, δε μπορεί να επαναχρησιμοποιηθεί για στοίβες άλλων τύπων
- Στην ML μπορούσαμε να χρησιμοποιήσουμε μεταβλητές τύπων για περιπτώσεις σαν και αυτές:

```
datatype 'a node =  
  NULL |  
  CELL of 'a * 'a node;
```

- Η Ada και η C++ έχουν κάτι παρόμοιο, αλλά όχι η Java

## Ζωή χωρίς γενικές κλάσεις στις Java 1.0-1.4

---

- Μπορούμε να ορίσουμε μια στοίβα της οποίας τα στοιχεία είναι αντικείμενα της κλάσης `Object`
- Ο τύπος `Object` είναι ο πιο γενικός τύπος της Java και περιλαμβάνει όλες τις αναφορές
- Κατά συνέπεια ο ορισμός μέσω της κλάσης `Object` επιτρέπει σε αντικείμενα οποιασδήποτε κλάσης να τοποθετηθούν στη στοίβα
- Το παραπάνω προσφέρει κάποιου είδους πολυμορφισμό υποτύπων

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

Κατά παρόμοιο τρόπο, θα μπορούσαμε να ορίσουμε την κλάση **GenericStack** (και μια **GenericWorklist** διαπροσωπεία) με χρήση **Object** στη θέση της **String**

# Μειονέκτημα

---

- Για να ανακτήσουμε τον τύπο του αντικειμένου στη στοίβα, θα πρέπει να χρησιμοποιήσουμε ένα **type cast**:

```
GenericStack s1 = new GenericStack();  
s1.add("hello");  
String s = (String) s1.remove();
```

- Το παραπάνω μάλλον δεν είναι ότι πιο φιλικό για τον προγραμματιστή
- Επίσης δεν είναι ότι πιο αποδοτικό σε χρόνο εκτέλεσης: η Java πρέπει να ελέγξει κατά τη διάρκεια εκτέλεσης του προγράμματος ότι το type cast επιτρέπεται – δηλαδή ότι το αντικείμενο είναι πράγματι ένα **String**

## Άλλο μειονέκτημα

---

- Οι πρωτόγονοι τύποι πρέπει πρώτα να αποθηκευθούν σε ένα αντικείμενο εάν θέλουμε να τους βάλουμε σε μια στοίβα:

```
GenericStack s2 = new GenericStack();  
s2.add(new Integer(42));  
int i = ((Integer) s2.remove()).intValue();
```

- Το παραπάνω είναι επίπονο και όχι ό,τι πιο αποδοτικό
- Η κλάση `Integer` είναι η προκαθορισμένη **κλάση περιτύλιγμα (wrapper class)** για τους ακεραίους
- Υπάρχει μια τέτοια κλάση για κάθε πρωτόγονο τύπο

## Πραγματικά Generics (Java 1.5, “Tiger”)

---

- Ξεκινώντας με τη Java 1.5, η Java έχει generics, δηλαδή παραμετρικές πολυμορφικές κλάσεις (και διαπροσωπείες)
- Η σύνταξή τους μοιάζει με τη σύνταξη των C++ templates

```
public class Stack<T> implements Worklist<T> {
    private Node<T> top = null;
    public void add(T data) {
        top = new Node<T>(data, top);
    }
    public boolean hasMore() {
        return (top != null);
    }
    public T remove() {
        Node<T> n = top;
        top = n.getLink();
        return n.getData();
    }
}
```

# Χρησιμοποίηση των Generics

---

```
Stack<String> s1 = new Stack<String>();  
Stack<int> s2 = new Stack<int>();  
s1.add("hello");  
String s = s1.remove();  
s2.add(42);  
int i = s2.remove();
```

```
class Stack {  
    void push(Object o) {  
        ... }  
    Object pop() { ... }  
    ...  
}  
String s = "Hello";  
Stack st = new Stack();  
...  
st.push(s);  
...  
s = (String) st.pop();
```

```
class Stack<T> {  
    void push(T a) { ... }  
    T pop() { ... }  
    ...  
}  
String s = "Hello";  
Stack<String> st =  
    new Stack<String>();  
st.push(s);  
...  
s = st.pop();
```

# Γιατί δεν υπήρξαν generics στις πρώτες Java;

---

- Υπήρξαν αρκετές προτάσεις για επέκταση
- Σε συμφωνία με τους βασικούς σκοπούς της γλώσσας
- Όμως “ο διάβολος είναι στις λεπτομέρειες”, όπως:
  - Τι παρενέργειες έχει η ύπαρξη generics για τη διαδικασία ελέγχου των τύπων;
  - Ποιος είναι ο καλύτερος τρόπος να γίνει η υλοποίηση;
    - Μπορεί η αφηρημένη μηχανή της Java να υποστηρίξει τα generics;
    - Αν ναι, μέσω πρόσθετων bytecodes ή με κάποιον άλλο τρόπο;
    - Μέσω ξεχωριστού κώδικα για κάθε στιγμιότυπο;
    - Ή μέσω του ίδιου κώδικα (με χρήση casts) για όλα τα στιγμιότυπα;

Το Java Community proposal (JSR 14) ενσωματώθηκε στη Java 1.5

## Generics στη Java 1.5 (“Tiger”)

---

- Υιοθετήθηκε η σύνταξη που μόλις είδαμε
- Προστέθηκε αυτόματη μετατροπή boxing + unboxing

Τι θα γράφαμε χωρίς αυτόματη μετατροπή

```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(new Integer(42));  
...  
int i = (st.pop()).intValue();
```

Τι γράφουμε στη Java 1.5

```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(42);  
...  
int i = st.pop();
```

# Οι τύποι των generics της Java ελέγχονται

---

- Μια γενική κλάση μπορεί να θελήσει να χρησιμοποιήσει λειτουργίες σε αντικείμενα ενός τύπου-παραμέτρου  
Παράδειγμα: `PriorityQueue<T> ... if x.less(y) then ...`
- Δύο πιθανές προσεγγίσεις:
  - C++: Κατά το χρόνο σύνδεσης (linking) ελέγχεται το κατά πόσο όλες οι λειτουργίες μπορούν να επιλυθούν
  - Java: Οι τύποι ελέγχονται στατικά και δε χρειάζεται να γίνει κάποιος έλεγχος των generics κατά το χρόνο σύνδεσης
    - Αυτή η προσέγγιση επιβάλλει στο πρόγραμμα να έχει πληροφορία για τον τύπο της παραμέτρου
    - Παράδειγμα : `PriorityQueue<T extends ...>`

## Παράδειγμα: Πίνακας κατακερματισμού

```
interface Hashable {
    int hashCode();
};
class HashTable <Key extends Hashable, Value> {
    void Insert(Key k, Value v) {
        int bucket = k.hashCode();
        InsertAt(bucket, k, v);
    }
    ...
};
```

Η έκφραση πρέπει να μην πετάει σφάλμα κατά τη διαδικασία ελέγχου των τύπων. Χρησιμοποιούμε “**Key extends Hashable**”

## Παράδειγμα: Ουρά προτεραιότητας

---

```
interface Comparable<I> {
    boolean lessThan(I);
};
class PriorityQueue<T> implements Comparable<T> {
    T queue[]; ...
    void insert(T t) {
        ... if (t.lessThan(queue[i]) ...
    }
    T remove() { ... }
    ...
};
```



## Ένα τελευταίο παράδειγμα ...

---

```
interface LessAndEqual<I> {
    boolean lessThan(I);
    boolean equal(I);
}
class Relations<C implements LessAndEqual<C>> extends C {
    boolean greaterThan(Relations<C> a) {
        return a.lessThan(this);
    }
    boolean greaterEqual(Relations<C> a) {
        return greaterThan(a) || equal(a);
    }
    boolean notEqual(Relations<C> a) { ... }
    boolean lessEqual(Relations<C> a) { ... }
    ...
}
```

# Υλοποίηση των Generics

---

- Διαγραφή τύπων (type erasure)
  - Ο έλεγχος τύπων κατά τη μετάφραση χρησιμοποιεί τα generics
  - Στη συνέχεια ο compiler απαλείφει τα generics μέσω διαγραφής
    - Δηλαδή μεταγλωττίζει `List<T>` σε `List`, `T` σε `Object`, και προσθέτει αυτόματες μετατροπές τύπων (casts)
- Τα generics της Java δεν είναι σαν τα templates της C++
  - Οι δηλώσεις των generics ελέγχονται ως προς τους τύπους τους
  - Τα generics μεταφράζονται άπαξ
    - Δεν λαμβάνει χώρα κάποια στιγμιοτυποποίηση (instantiation)
    - Ο παραγόμενος κώδικας δεν διογκώνεται