

1. ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ

Προγραμματιστικά παραδείγματα.

Εισαγωγή στις βασικές έννοιες του αντικειμενοστραφούς προγραμματισμού

(Ευχαριστίες στον καθηγητή Βασίλη Χριστοφίδη)

Λίγο Ιστορία

- Οι πρώτες γλώσσες προγραμματισμού δεν ήταν για υπολογιστές
 - Αυτόματη δημιουργία πρωτοτύπων για ραπτομηχανές
 - Μουσικά κουτιά ή ρολά για πιάνο
 - Η αφαιρετική μηχανή του Turing

Γλώσσες προγραμματισμού

- **Πρώτη γενιά:** Γλώσσες μηχανής

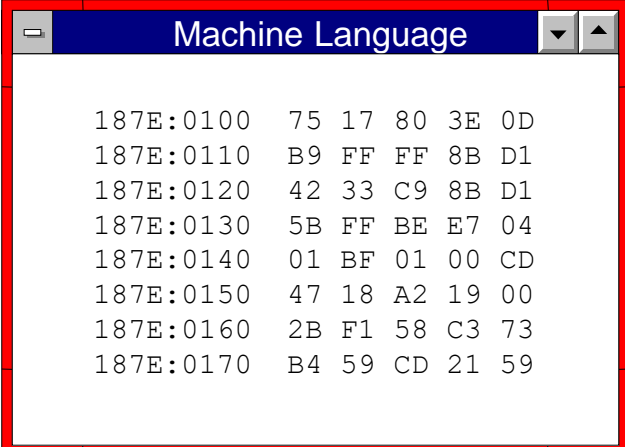
Ο προγραμματιστής μετατρέπει το πρόβλημα του σε ένα πρόγραμμα

- Π.χ. πώς να υπολογίσω το μέγιστο κοινό διαιρέτη δύο αριθμών

Και γράφει **ακριβώς** τις εντολές που θα πρέπει να εκτελέσει ο υπολογιστής

- Θα πρέπει να ξέρει ακριβώς την δυαδική αναπαράσταση των εντολών.

Στους πρώτους υπολογιστές οι εντολές κωδικοποιούνταν σε διάτρητες κάρτες



```
Machine Language
187E:0100  75 17 80 3E 0D
187E:0110  B9 FF FF 8B D1
187E:0120  42 33 C9 8B D1
187E:0130  5B FF BE E7 04
187E:0140  01 BF 01 00 CD
187E:0150  47 18 A2 19 00
187E:0160  2B F1 58 C3 73
187E:0170  B4 59 CD 21 59
```

Program entered and executed as machine language

Πέντε γενεές γλωσσών προγραμματισμού

- Πρώτη γενιά: Γλώσσες μηχανής
- Δεύτερη γενιά: Assembly

The ASSEMBLER converts instructions to op-codes:
What is the instruction to load from memory?
Where is purchase price stored?
What is the instruction to multiply?
What do I multiply by?
What is the instruction to add from memory?
What is the instruction to store back into memory?

Ο προγραμματιστής δεν χρειάζεται να ξέρει ακριβώς την δυαδική αναπαράσταση των εντολών.

- Χρησιμοποιεί πιο κατανοητούς **μνημονικούς κανόνες**.
- Ο **Assembler** μετατρέπει τα σύμβολα σε γλώσσα μηχανής.
- Οι γλώσσες **εξαρτώνται** από το **hardware**

```
Assembly Language
POP  SI
MOV  AX, [BX+03]
SUB  AX, SI
MOV  WORD PTR [TOT_AMT], E0D7
MOV  WORD PTR [CUR_AMT], E1DB
ADD  [TOT_AMT], AX
```

Translate into machine operation codes (op-codes)

```
Machine Language
187E:0100  75 17 80 3E 0D
187E:0110  B9 FF FF 8B D1
187E:0120  42 33 C9 8B D1
187E:0130  5B FF BE E7 04
187E:0140  01 BF 01 00 CD
187E:0150  47 18 A2 19 00
187E:0160  2B F1 58 C3 73
187E:0170  B4 59 CD 21 59
```

Program executed as machine language

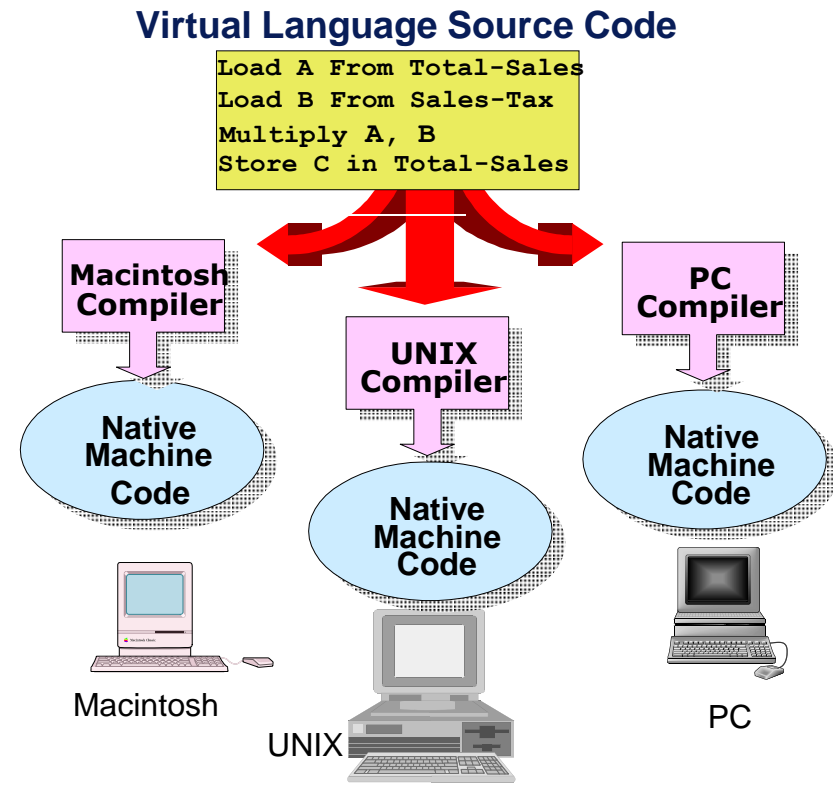
Πέντε γενεές γλωσσών προγραμματισμού

- **Πρώτη γενιά:** Γλώσσες μηχανής
- **Δεύτερη γενιά:** Assembly
- **Τρίτη γενιά:** Υψηλού επιπέδου (high-level) γλώσσες

Ο προγραμματιστής δίνει εντολές στον υπολογιστή σε μια κατανοητή και καλά δομημένη **γλώσσα (source code)**

Ο **compiler** τις μετατρέπει σε **ενδιάμεσο κώδικα (object code)**

Ο ενδιάμεσος κώδικας μετατρέπεται σε **γλώσσα μηχανής (machine code)**



Πέντε γενεές γλωσσών προγραμματισμού

- Πρώτη γενιά: Γλώσσες μηχανής
- Δεύτερη γενιά: Assembly
- Τρίτη γενιά: Υψηλού επιπέδου (high-level) γλώσσες

The **COMPILER** translates:
Load the purchase price
Multiply it by the sales tax
Add the purchase price to the result
Store the result in total price

```
High-Level Language
```

```
-  
salesTax = purchasePric * TAX_RATE;  
totalSales = purchasePrice + salesTax;
```

Translate into the instruction set

```
Assembly Language
```

```
POP SI  
MOV AX, [BX+03]  
SUB AX, SI  
MOV WORD PTR [TOT_AMT], E0D7  
MOV WORD PTR [CUR_AMT], E1DE  
ADD [TOT_AMT], AX
```

Translate into machine operation codes (op-codes)

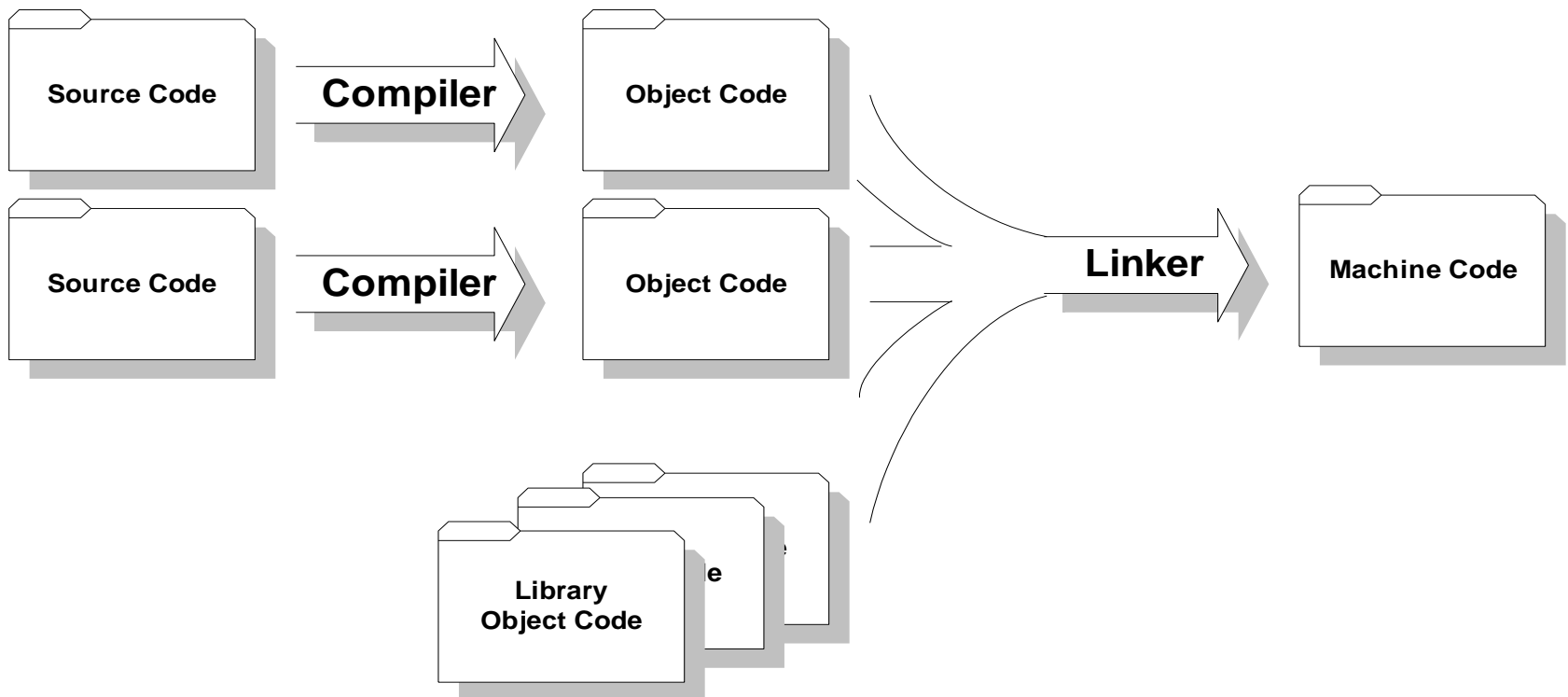
```
Machine Language
```

```
1  
87E:0100 75 17 80 3E 0D  
87E:0110 B9 FF FF 8B D1  
87E:0120 42 33 C9 8B D1  
87E:0130 5B FF BE E7 04  
87E:0140 01 BF 01 00 CD  
87E:0150 47 18 A2 19 00  
87E:0160 2B F1 58 C3 73  
87E:0170 B4 59 CD 21 59
```

Program executed as machine language

Πέντε γενεές γλωσσών προγραμματισμού

- **Πρώτη γενιά:** Γλώσσες μηχανής
- **Δεύτερη γενιά:** Assembly
- **Τρίτη γενιά:** Υψηλού επιπέδου (high-level) γλώσσες



Πέντε γενεές γλωσσών προγραμματισμού

- **Πρώτη γενιά:** Γλώσσες μηχανής
 - **Δεύτερη γενιά:** Assembly
 - **Τρίτη γενιά:** Υψηλού επιπέδου (high-level) γλώσσες
 - **Τέταρτη γενιά:** Εξειδικευμένες γλώσσες
 - **Πέμπτη γενιά:** «Φυσικές» γλώσσες.
-
- Κάθε γενιά προσθέτει ένα επίπεδο **αφαίρεσης**.

Προγραμματιστικά Υποδείγματα (paradigms)

- Προγραμματισμός των πρώτων ημερών.

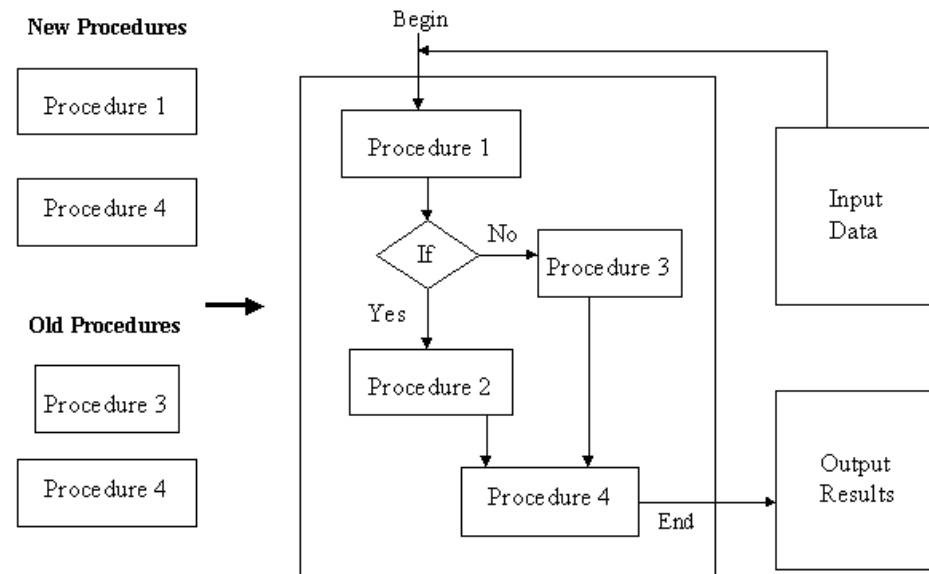
Spaghetti code

Δύσκολο να διαβαστεί και να κατανοηθεί η ροή του

```
100  GOTO 500
110  PRINT I;
120  GOTO 400
150  PRINT I * 12;
160  GOTO 450
200  PRINT " = ";
210  GOTO 150
300  PRINT " 12 ";
310  GOTO 200
400  PRINT " * ";
410  GOTO 300
450  I = I + 1
460  IF I > 12 THEN STOP
460  GOTO 110
500  PRINT "The Twelves Table"
510  I = 1
520  GOTO 110
```

Δομημένος Προγραμματισμός

- Τέσσερις προγραμματιστικές **δομές**
 - **Sequence** – ακολουθιακές εντολές
 - **Selection** – επιλογή με if-then-else
 - **Iteration** – δημιουργία βρόγχων
 - **Recursion** - αναδρομή
- Ο κώδικας σπάει σε λογικά **blocks** που έχουν **ένα σημείο εισόδου** και **εξόδου**.
 - **Κατάργηση** της **GOTO** εντολής.
- Οργάνωση του κώδικα σε **διαδικασίες (procedures)**

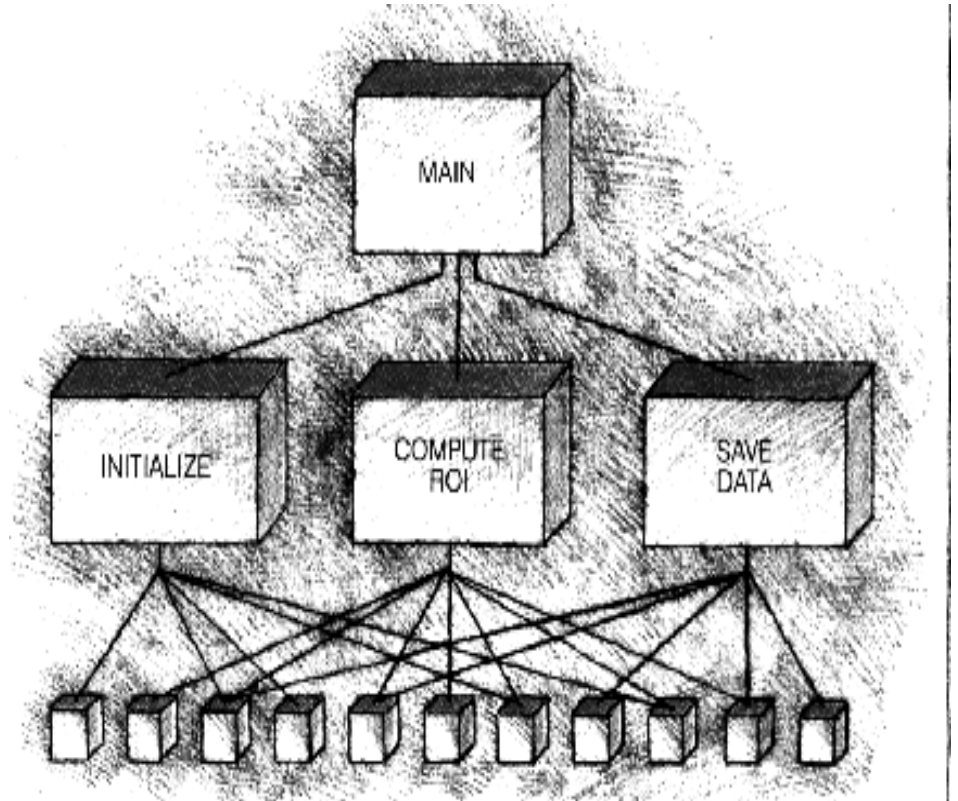


Διαδικασιακός Προγραμματισμός

- Το πρόγραμμα μας σπάει σε πολλαπλές διαδικασίες.
 - Κάθε διαδικασία λύνει ένα υπο-πρόβλημα και αποτελεί μια λογική μονάδα (**module**)
 - Μια διαδικασία μπορούμε να την επαναχρησιμοποιήσουμε σε διαφορετικά δεδομένα.
- Το πρόγραμμα μας είναι **τμηματοποιημένο** (**modular**)

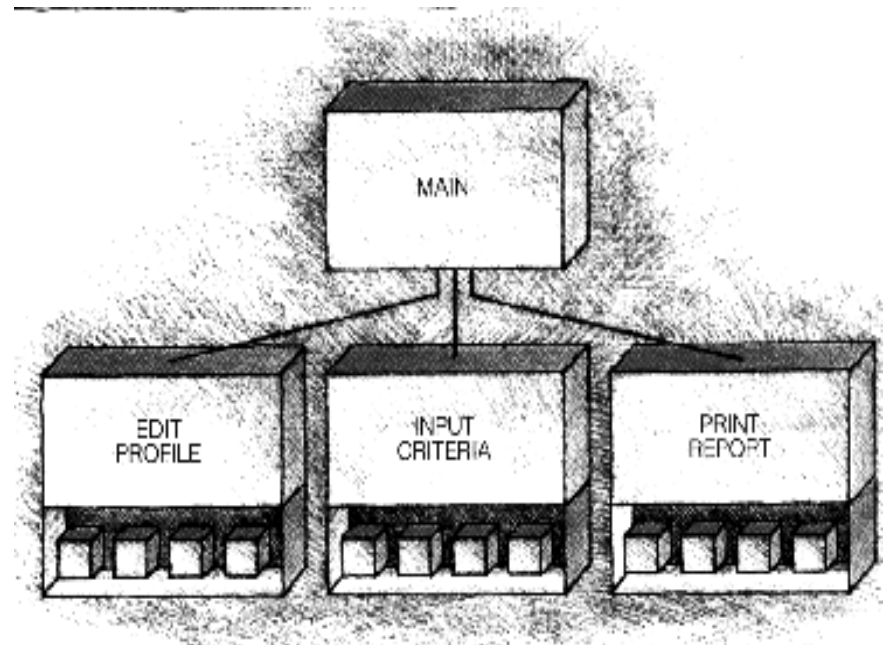
Κοινά Δεδομένα

- Ο διαδικασιακός προγραμματισμός τμηματοποιεί τον κώδικα αλλά **όχι** απαραίτητα **τα δεδομένα**
- Π.χ., με τη χρήση **καθολικών μεταβλητών** (global variables) όλες οι διαδικασίες μπορεί να χρησιμοποιούν τα ίδια δεδομένα και άρα να εξαρτώνται μεταξύ τους.
- Πρέπει να **αποφεύγουμε** τη **χρήση καθολικών μεταβλητών!**



Απόκρυψη δεδομένων

- Με τη δημιουργία **τοπικών μεταβλητών** μέσα στις διαδικασίες αποφεύγουμε την ύπαρξη κοινών δεδομένων
- Ο κώδικας γίνεται πιο εύκολο να σχεδιαστεί, να γραφτεί και να συντηρηθεί
- Η επικοινωνία μεταξύ των διαδικασιών γίνεται με **ορίσματα**.
- **Τμηματοποιημένος προγραμματισμός** (**modular programming**)

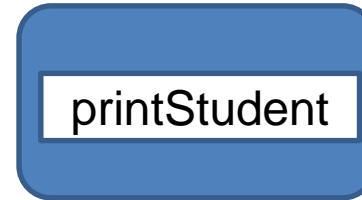
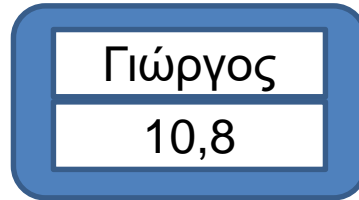


Περιορισμοί του διαδικασιακού προγραμματισμού

- Ο διαδικασιακός προγραμματισμός δουλεύει OK για μικρά προγράμματα, αλλά για μεγάλα συστήματα είναι δύσκολο να **σχεδιάσουμε**, να **υλοποιήσουμε** και να **συντηρήσουμε** τον κώδικα.
 - Δεν είναι εύκολο να προσαρμοστούμε σε αλλαγές, και δεν μπορούμε να προβλέψουμε όλες τις ανάγκες που θα έχουμε
- Π.χ., το πανεπιστήμιο έχει ένα σύστημα για να κρατάει πληροφορίες για φοιτητές και καθηγητές
 - Υπάρχει μια διαδικασία **printStudent** που τυπώνει στοιχεία και **βαθμούς φοιτητών**
 - Προκύπτει ανάγκη για μια διαδικασία που να τυπώνει τα **μαθήματα των καθηγητών**
 - Χρειαζόμαστε μια **printProf**

Παράδειγμα

Φοιτητής studentGeorge:



Καθηγητής profKostas:

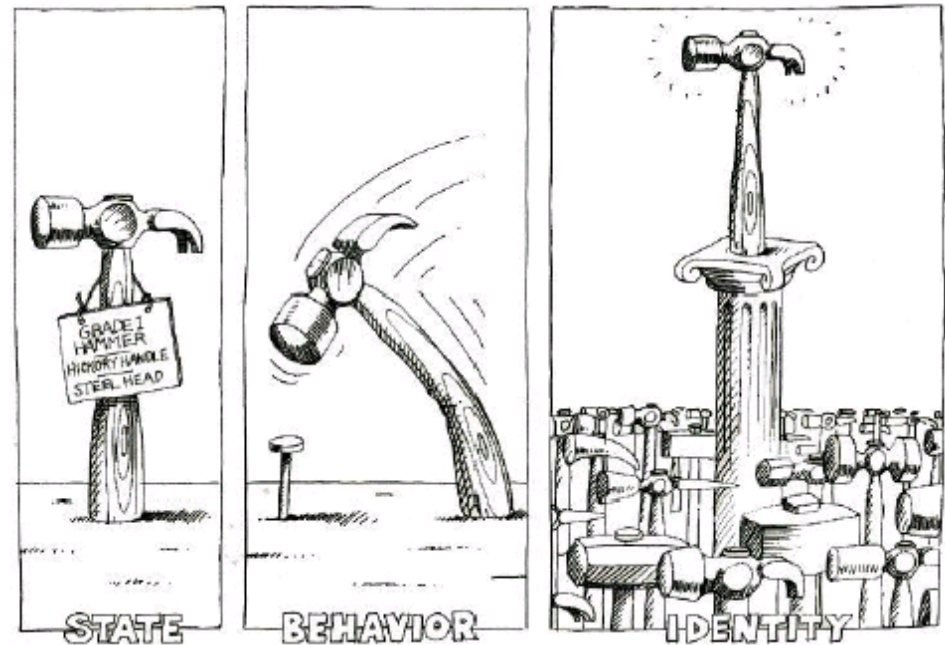


Αντικειμενοστραφής προγραμματισμός

- Τα προβλήματα αυτά προσπαθεί να αντιμετωπίσει ο αντικειμενοστραφής προγραμματισμός (object-oriented programming)
 - Ο αντικειμενοστραφής προγραμματισμός βάζει **μαζί** τα **δεδομένα** και τις **διαδικασίες** (μεθόδους) σχετικές με τα δεδομένα
 - Π.χ., ο φοιτητής ή ο καθηγητής έρχεται με μια δικιά του διαδικασία print
- Αυτό επιτυγχάνεται με **αντικείμενα** και **κλάσεις**

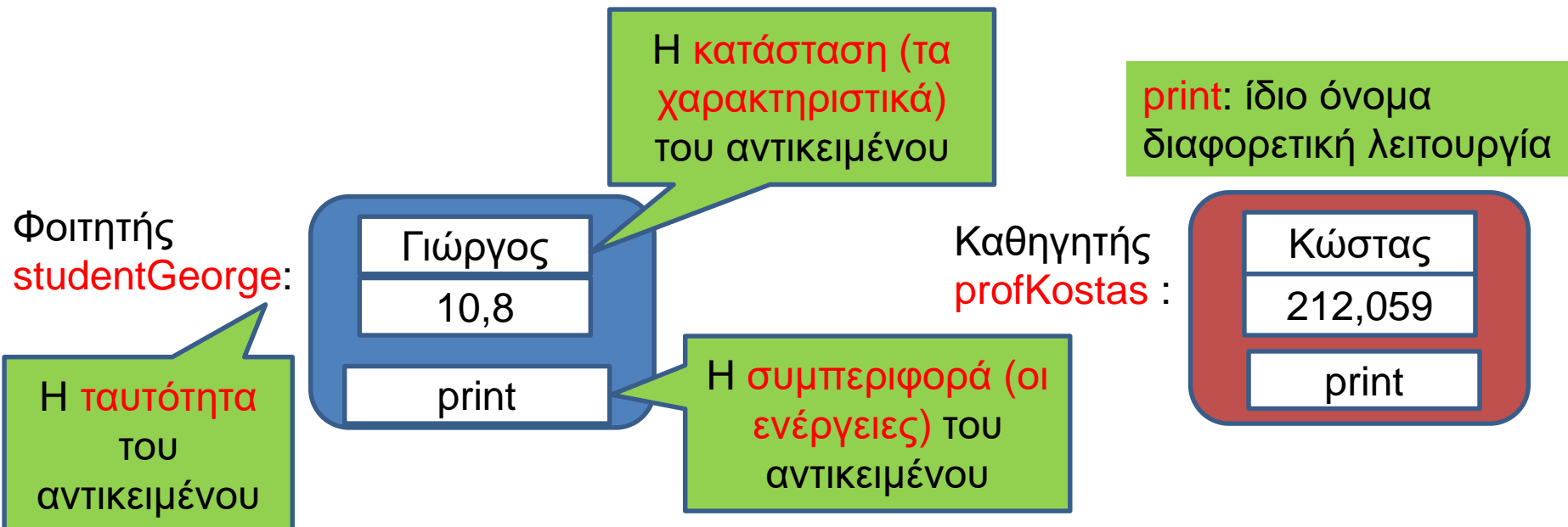
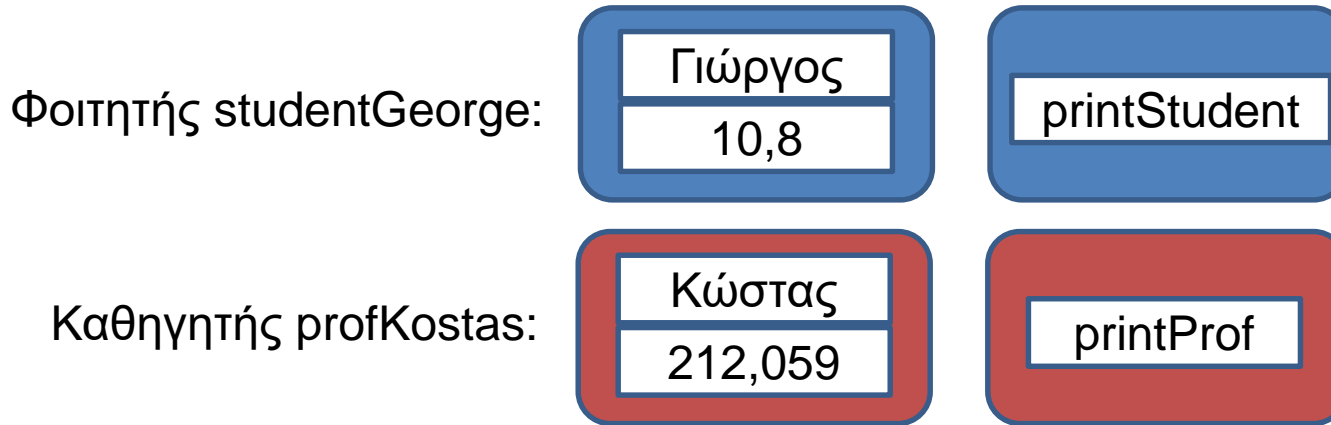
Αντικείμενο

- Ένα αντικείμενο στον κώδικα αναπαριστά μια μονάδα/οντότητα/έννοια η οποία έχει:
 - Μια **κατάσταση**, η οποία ορίζεται από ορισμένα **χαρακτηριστικά**
 - Μια **συμπεριφορά**, η οποία ορίζεται από ορισμένες **ενέργειες** που μπορεί να εκτελέσει το αντικείμενο
 - Μια **ταυτότητα** που το ξεχωρίζει από τα υπόλοιπα αντικείμενα **ίδιου τύπου**.



Παραδείγματα: ένας άνθρωπος, ένα πράγμα, ένα μέρος, μια υπηρεσία

Παράδειγμα



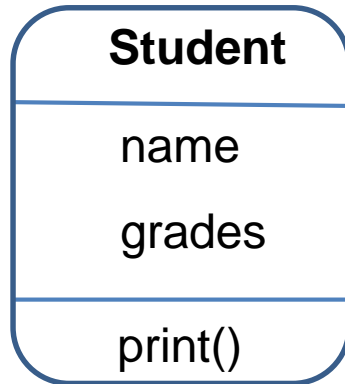
Κλάσεις

- **Κλάση**: Μια αφηρημένη περιγραφή αντικειμένων με κοινά χαρακτηριστικά και κοινή συμπεριφορά
 - Ένα καλούπι που παράγει αντικείμενα
 - Ένα αντικείμενο είναι ένα **στιγμιότυπο** μίας κλάσης.
- Π.χ., η κλάση **φοιτητής** έχει τα γενικά χαρακτηριστικά (όνομα, βαθμοί) και τη συμπεριφορά `print`
 - Ο φοιτητής X είναι ένα **αντικείμενο** της **κλάσης** φοιτητής
- Η **κλάση Car** έχει τα χαρακτηριστικά (**brand, color**) και τη συμπεριφορά (**drive, stop**)
 - Το αυτοκίνητο **INI2013** είναι ένα **αντικείμενο** της κλάσης Car με κατάσταση τα χαρακτηριστικά (**BMW, red**)

Κλάσεις και Αντικείμενα

Κλάση

Μια αφηρημένη περιγραφή ενός φοιτητή.



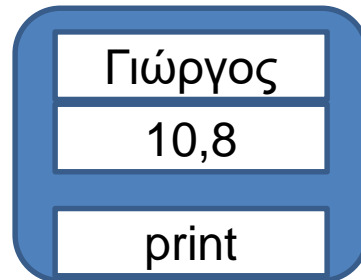
Όνομα κλάσης

Πεδία κλάσης: Ιδιότητες/Χαρακτηριστικά

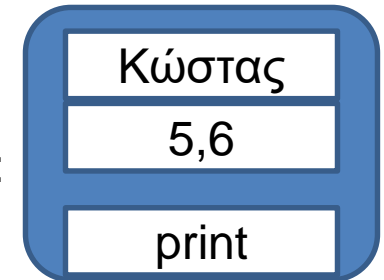
Μέθοδοι κλάσης: λειτουργίες

Αντικείμενα

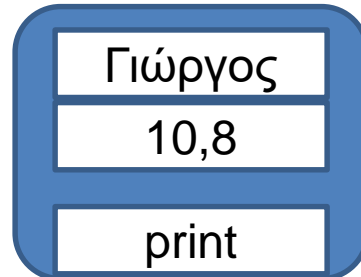
Student
studentGeorge1:



Student
studentKostas:



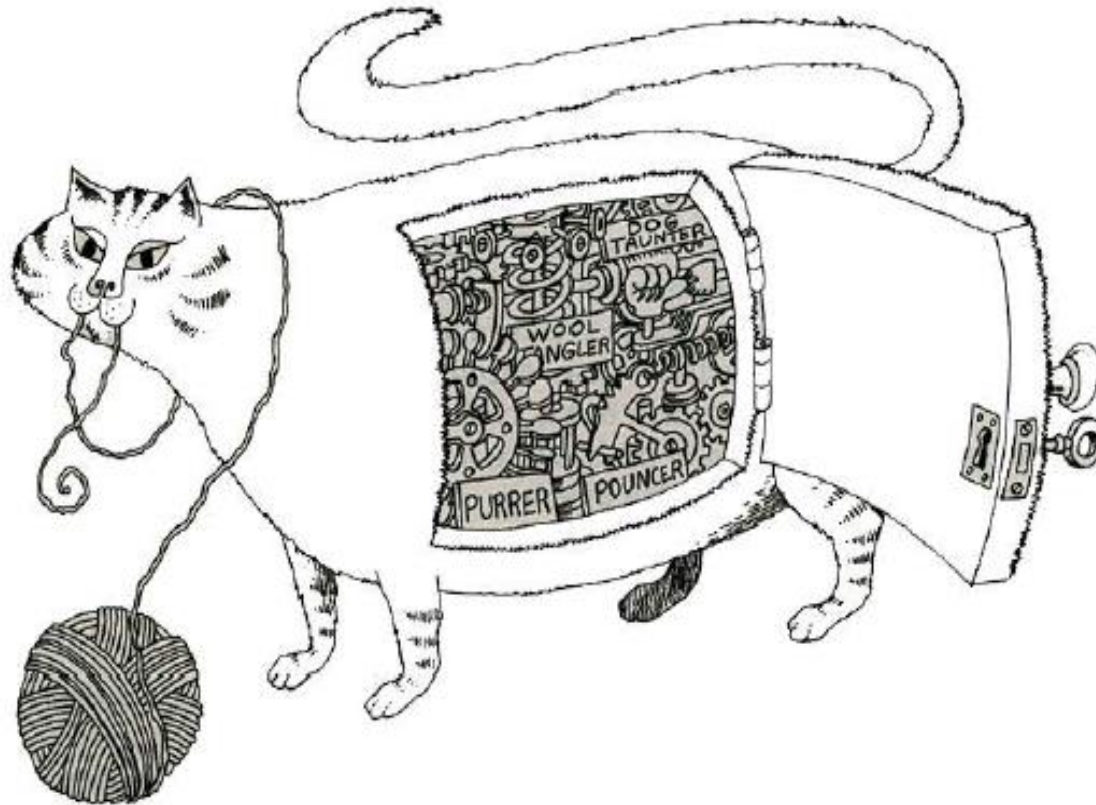
Student
studentGeorge2:



Το κάθε αντικείμενο έχει

- Μια κατάσταση (το συγκεκριμένο όνομα και τους συγκεκριμένους βαθμούς)
- Ενέργειες (τις λειτουργίες/μεθόδους)
- Ταυτότητα (studentGeorge1, studentGeorge2, studentKostas)

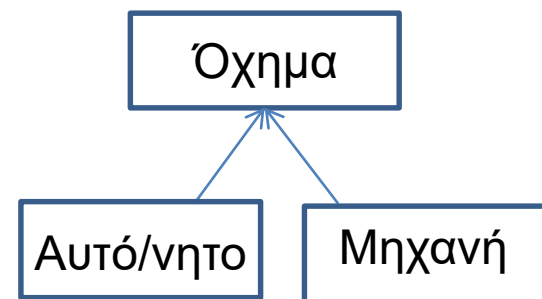
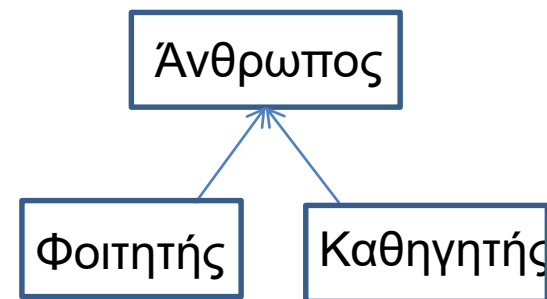
Ενθυλάκωση



- Η στεγανοποίηση της κατάστασης και της συμπεριφοράς ώστε οι λεπτομέρειες της υλοποίησης να είναι κρυμμένες από το χρήστη του αντικειμένου.

Κληρονομικότητα

- Οι κλάσεις μας επιτρέπουν να ορίσουμε μια **ιεραρχία**
 - Π.χ., και ο **Φοιτητής** και ο **Καθηγητής** ανήκουν στην κλάση **Άνθρωπος**.
 - Η κλάση **Αυτοκίνητο** ανήκει στην κλάση **Όχημα** η οποία περιέχει και την κλάση **Μοτοσυκλέτα**
- Οι κλάσεις πιο χαμηλά στην ιεραρχία **κληρονομούν** χαρακτηριστικά και συμπεριφορά από τις ανώτερες κλάσεις
 - Όλοι οι άνθρωποι έχουν **όνομα**
 - Όλα τα οχήματα έχουν μέθοδο **drive**, **stop**.



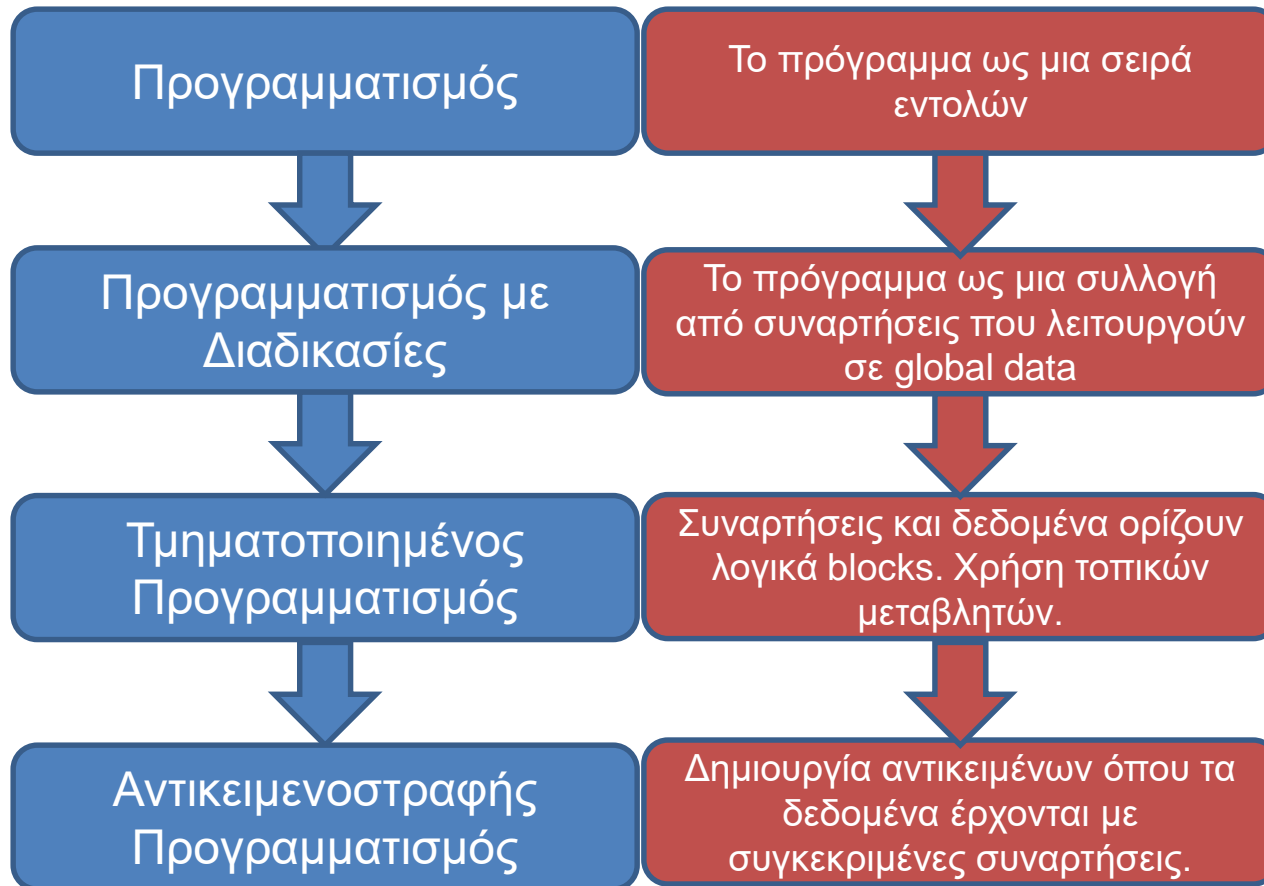
Πολυμορφισμός

- Κλάσεις με κοινό πρόγονο έχουν κοινά χαρακτηριστικά, αλλά έχουν και διαφορές
 - Π.χ., είναι διαφορετικό το **παρκάρισμα** για ένα αυτοκίνητο και μια μηχανή
- Ο **πολυμορφισμός** μας επιτρέπει να δώσουμε μια **κοινή** συμπεριφορά σε κάθε κλάση (μια μέθοδο **park**), η οποία όμως **υλοποιείται διαφορετικά** για αντικείμενα διαφορετικών κλάσεων.
- Μπορούμε επίσης να ορίσουμε **αφηρημένες κλάσεις**, όπου **προϋποθέτουμε** μια συμπεριφορά και αυτή πρέπει να υλοποιηθεί σε χαμηλότερες κλάσεις διαφορετικά ανάλογα με τις ανάγκες μας

Αφηρημένοι Τύποι Δεδομένων

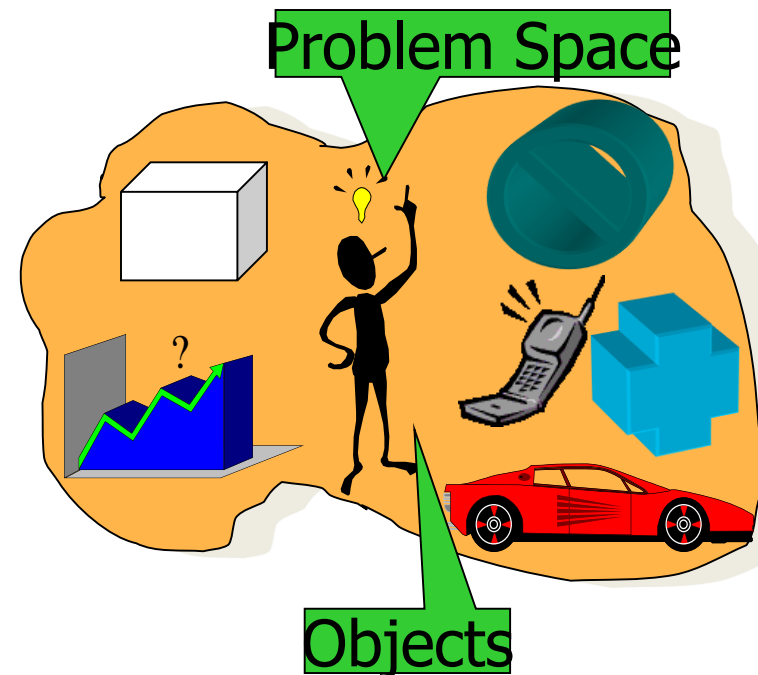
- Χρησιμοποιώντας τις κλάσεις μπορούμε να ορίσουμε τους δικούς μας **τύπους δεδομένων**
 - Έτσι μπορούμε να φτιάξουμε αντικείμενα με συγκεκριμένα χαρακτηριστικά και συμπεριφορά.
- Χρησιμοποιώντας την κληρονομικότητα και τον πολυμορφισμό, μπορούμε να **επαναχρησιμοποιήσουμε** υπάρχοντα χαρακτηριστικά και μεθόδους.

Η εξέλιξη του προγραμματισμού



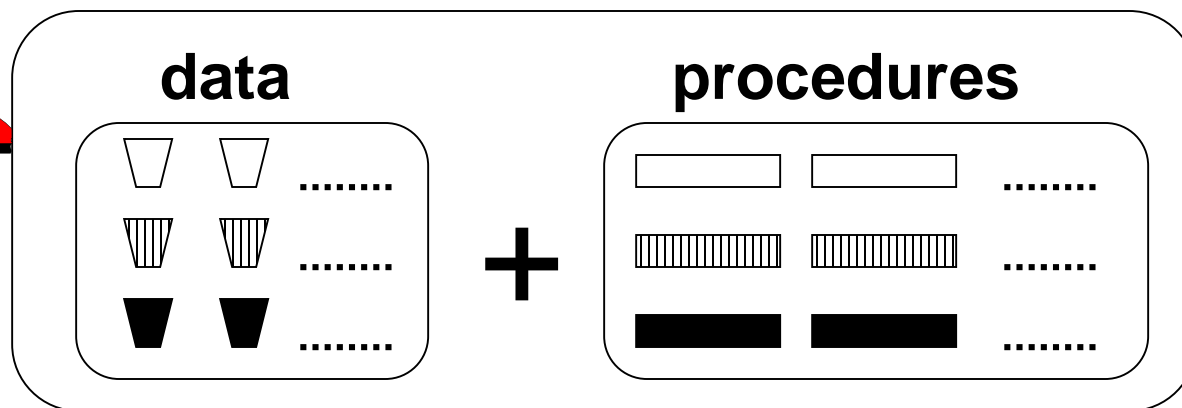
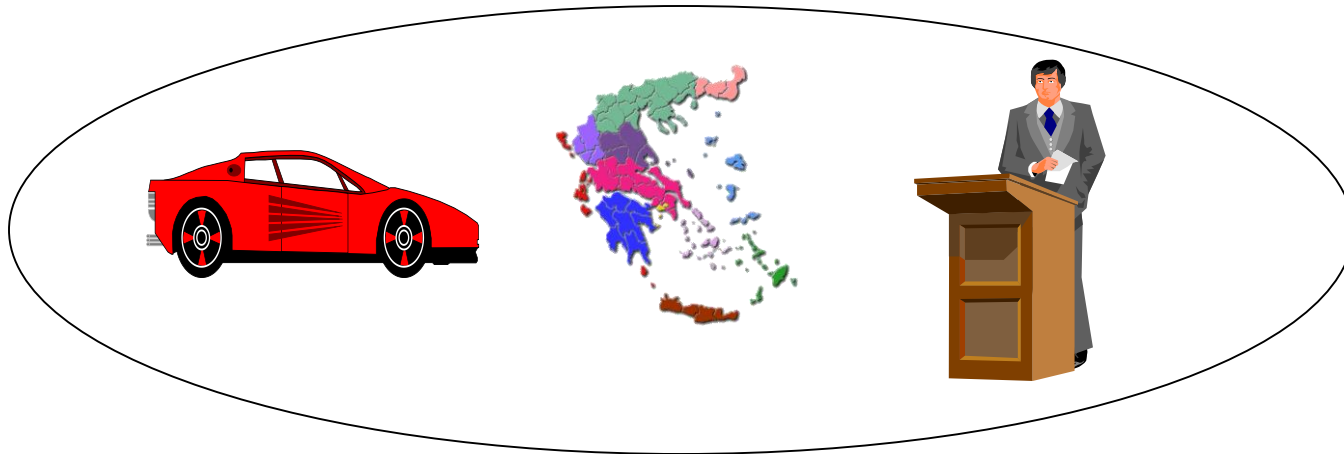
Διαδικασιακός vs. Αντικειμενοστραφής Προγραμματισμός

- **Διαδικασιακός:** Έμφαση στις διαδικασίες
 - Οι δομές που δημιουργούμε είναι για να ταιριάζουν με τις διαδικασίες.
 - Οι διαδικασίες προκύπτουν από το χώρο των λύσεων.
- **Αντικειμενοστραφής:** Έμφαση στα αντικείμενα
 - Τα αντικείμενα δημιουργούνται από το χώρο του προβλήματος
 - Λειτουργούν ακόμη και αν αλλάξει το πρόβλημα μας



Διαδικασιακή αναπαράσταση

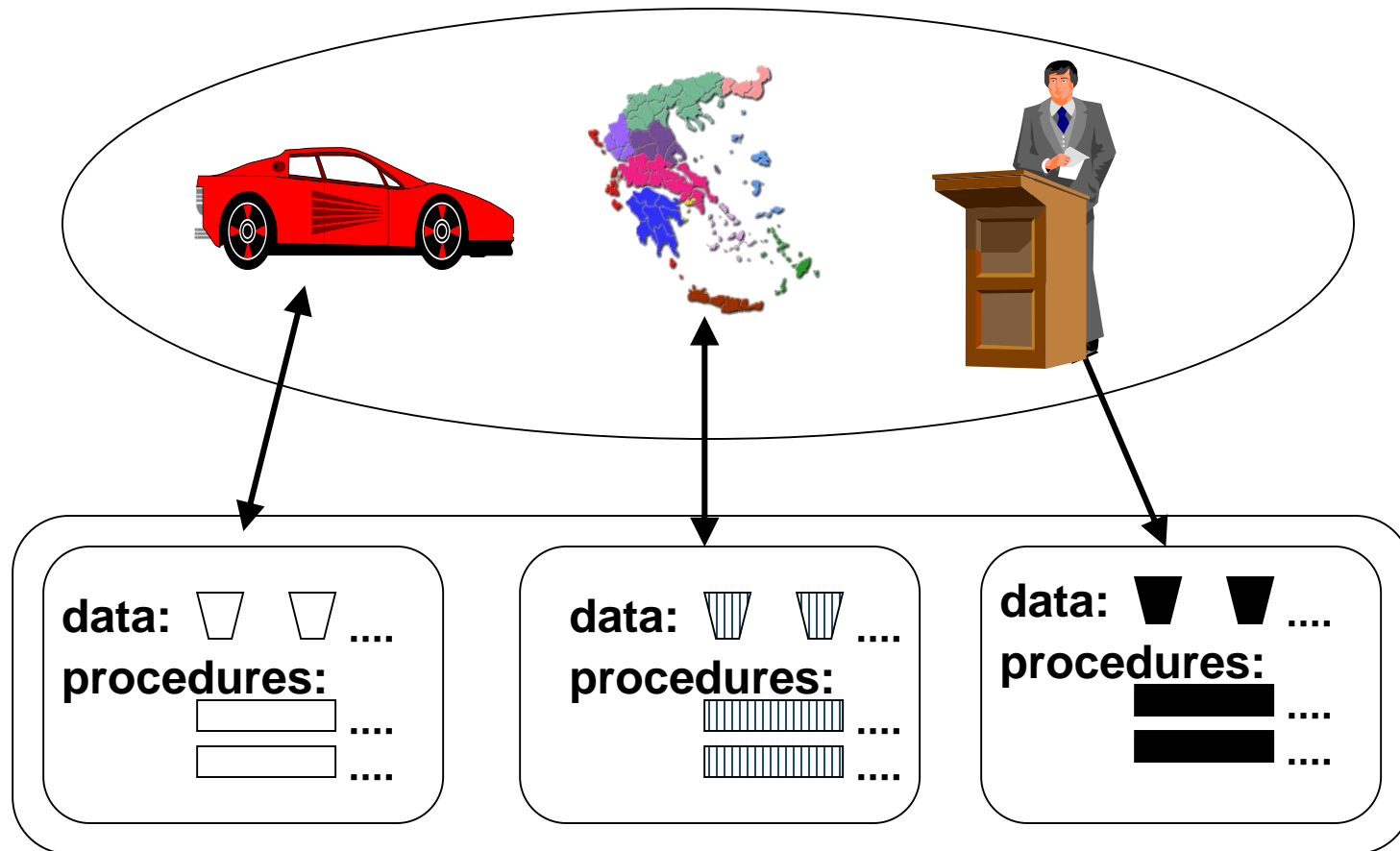
Real world entities



Software Representation

Αντικειμενοστραφής αναπαράσταση

Real world entities

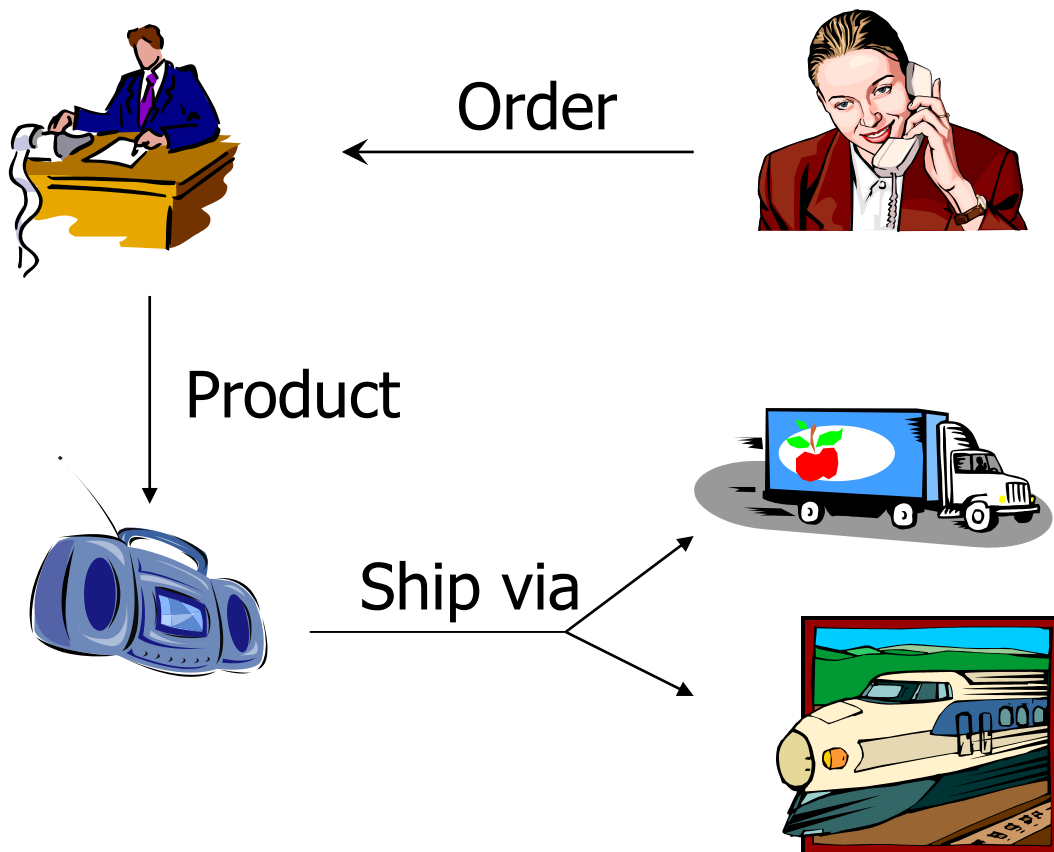


Software Representation

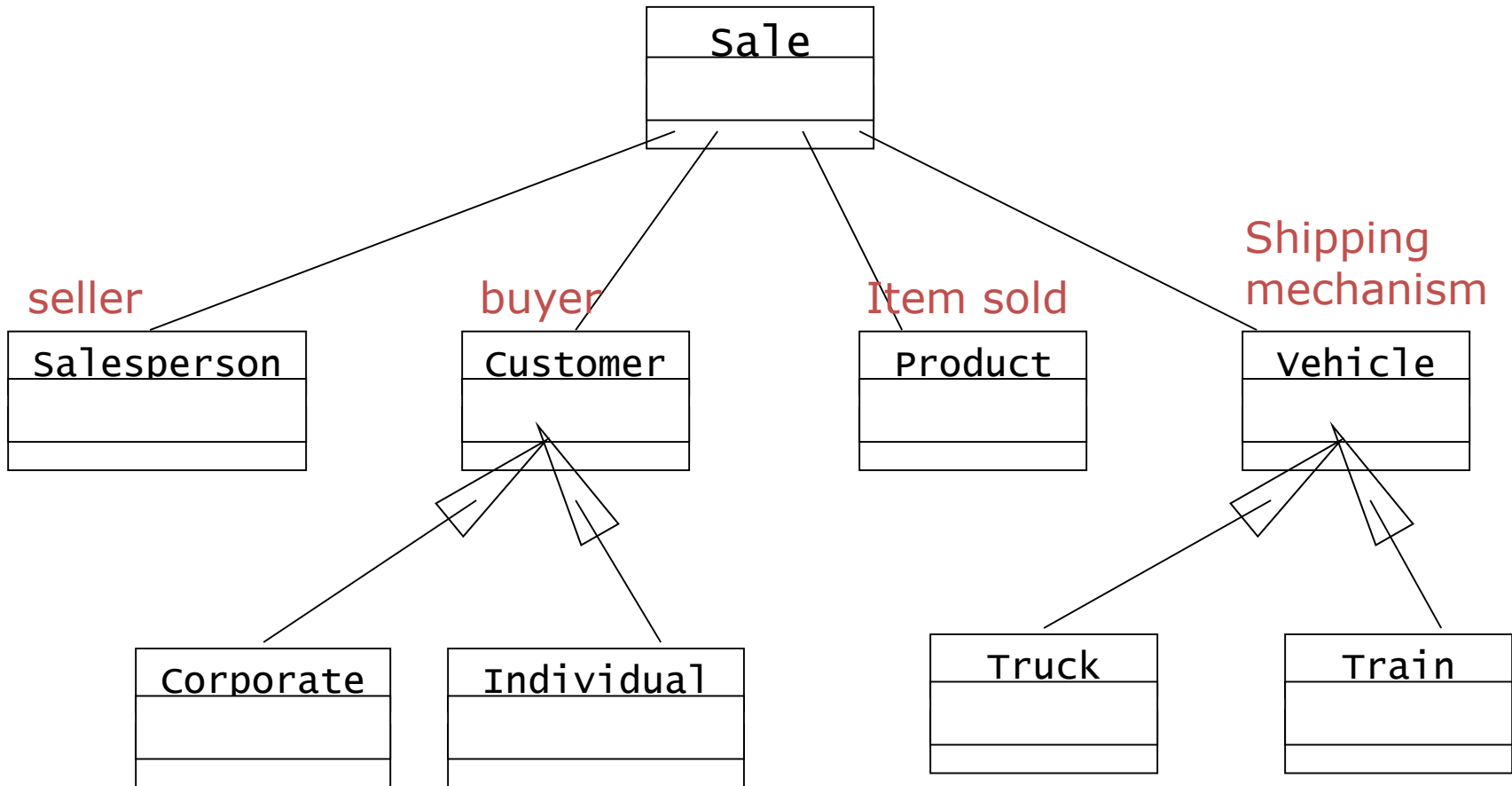
Παράδειγμα: Πωλήσεις

Θέλουμε να δημιουργήσουμε λειτουργικό για ένα σύστημα το οποίο διαχειρίζεται πωλήσεις.

- Πελάτες **κάνουν** παραγγελίες.
- Οι πωλητές **χειρίζονται** την παραγγελία
- Οι παραγγελίες είναι για συγκεκριμένα **προϊόντα**
- Η παραγγελία **αποστέλλεται** με επιλεγμένο **μέσο**

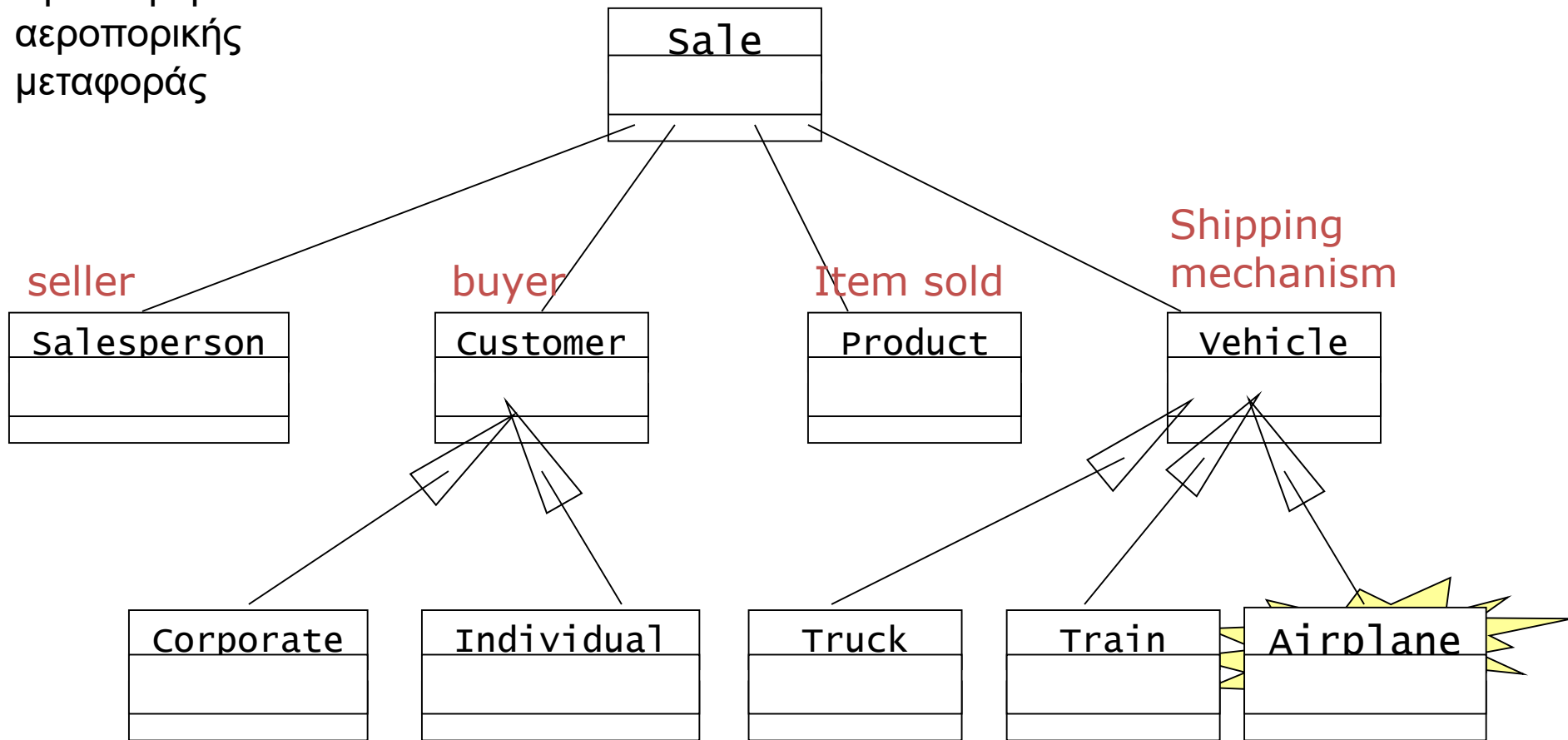


Διάγραμμα κλάσεων



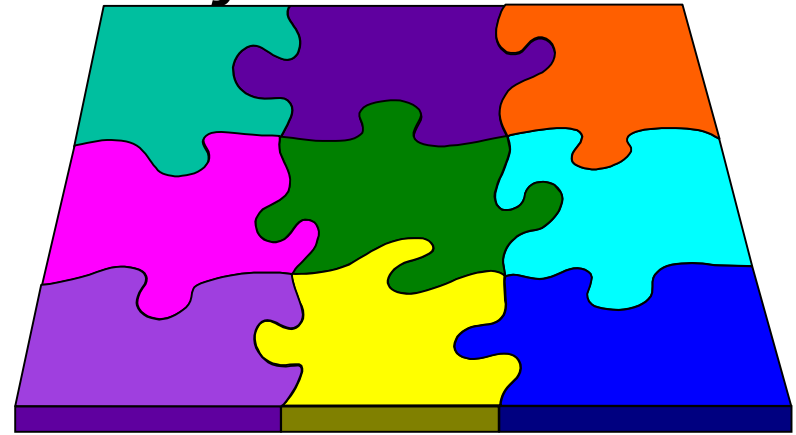
Αλλαγή των απαιτήσεων

Προσθήκη
αεροπορικής
μεταφοράς



Πλεονεκτήματα αντικειμενοστραφούς προγραμματισμού

- Επειδή προσπαθεί να μοντελοποιήσει τον πραγματικό κόσμο, ο OOP κώδικας είναι πιο κατανοητός.
- Τα δομικά κομμάτια που δημιουργεί είναι πιο εύκολο να επαναχρησιμοποιηθούν και να συνδυαστούν
- Ο κώδικας είναι πιο εύκολο να συντηρηθεί λόγω της ενθυλάκωσης



2. ΕΙΣΑΓΩΓΗ ΣΤΗ JAVA

Η γλώσσα προγραμματισμού Java

Βασικό συντακτικό, ορισμός μεταβλητών,
έλεγχος ροής

Η εξέλιξη των γλωσσών προγραμματισμού

- Η εξέλιξη των γλωσσών προγραμματισμού είναι μια διαδικασία **αφαίρεσης**
 - Στην αρχή ένα πρόγραμμα ήταν μια σειρά από εντολές σε γλώσσα μηχανής.
 - Με τον **Διαδικασιακό Προγραμματισμό (procedural programming)**, ένα πρόγραμμα έγινε μια συλλογή από **διαδικασίες** που η μία καλεί την άλλη.
 - Στον **Συναρτησιακό Προγραμματισμό (functional programming)** ένα πρόγραμμα είναι μια συλλογή από **συναρτήσεις** όπου η μία εφαρμόζεται πάνω στην άλλη.
 - Στον **Λογικό Προγραμματισμό (logic programming)** ένα πρόγραμμα είναι μια συλλογή από **κανόνες** και **γεγονότα**.
 - Στον **Αντικειμενοστραφή Προγραμματισμό (object oriented programming)** ένα πρόγραμμα είναι μια συλλογή από **κλάσεις** και **αντικείμενα** όπου το ένα μιλάει με το άλλο

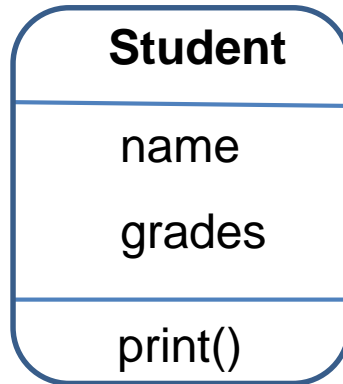
Αντικειμενοστραφής Προγραμματισμός

- Οι πέντε αρχές του Allan Kay:
 - Τα πάντα είναι **αντικείμενα**.
 - Ένα πρόγραμμα είναι μια **συλλογή** από **αντικείμενα** όπου το ένα λέει στο άλλο τι να κάνει.
 - Κάθε αντικείμενο έχει δικιά του **μνήμη** και αποτελείται από **άλλα αντικείμενα**.
 - Κάθε αντικείμενο έχει ένα συγκεκριμένο **τύπο**.
 - Τύπος = **Κλάση**
 - Αντικείμενα του **ίδιου τύπου** μπορούν να δεχτούν **τα ίδια μηνύματα**
 - Δηλαδή έχουν τις **ίδιες λειτουργίες**

Κλάσεις και Αντικείμενα

Κλάση

Μια αφηρημένη περιγραφή μιας οντότητας



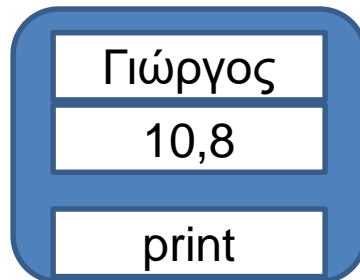
Όνομα κλάσης

Πεδία κλάσης: Ιδιότητες/Χαρακτηριστικά

Μέθοδοι κλάσης: λειτουργίες

Αντικείμενα

Student **studentGeorge**:



Ένα συγκεκριμένο στιγμιότυπο της αφηρημένης κλάσης

Πρόσβαση στο αντικείμενο μόνο μέσω κλήσεων των μεθόδων:

```
studentGeorge.print()
```

Τυπώνει τις πληροφορίες για το αντικείμενο.

Σύντομη ιστορία του Αντικειμενοστραφούς Προγραμματισμού

- Η πρώτη γλώσσα που χρησιμοποίησε τις έννοιες της κλάσης και του αντικειμένου θεωρείται η **SIMULA** (1960s)
 - Γλώσσα για προσομοιώσεις συστημάτων
- Εμπνευσμένος από την SIMULA ο **Allan Kay** δημιούργησε στην HP την γλώσσα **SmallTalk** με στόχο μια γλώσσα που να υποστηρίζει γραφικά (1970s)
 - Ήταν αυτός που εισήγαγε την έννοια «**Αντικειμενοστραφής Προγραμματισμός**» (**Object Oriented Programming**)
 - Το 2003 βραβεύτηκε με το Turing Award
- Οι ιδέες του αντικειμενοστραφούς προγραμματισμού άρχισαν να εισάγονται σε πολλές υπάρχουσες ή νέες γλώσσες. Ο **Bjorn Stroustrup** δημιούργησε την **C++** (1980s)
- Η Sun δημιούργησε την γλώσσα **Java** η οποία βρίσκει εφαρμογή σε ανάπτυξη εφαρμογών στο διαδίκτυο (1990s)
 - Ακολούθησε η Microsoft με την .NET πλατφόρμα και τις γλώσσες **Visual Basic** και **C#**

Ιστορία της Java

- Ο **Patrick Naughton** απειλεί την Sun ότι θα φύγει.
- Τον βάζουν σε μία ομάδα αποτελούμενη από τους **James Gosling** και **Mike Sheridan** για να σχεδιάσουν τον προγραμματισμό των έξυπνων συσκευών της επόμενης γενιάς.
 - The **Green project**.
- Ο Gosling συνειδητοποιεί ότι η C++ δεν είναι αρκετά αξιόπιστη για να δουλεύει σε συσκευές περιορισμένων δυνατοτήτων και με διάφορες αρχιτεκτονικές.
 - Δημιουργεί τη γλώσσα **Oak**
- Το 1992 η ομάδα κάνει ένα demo μιας συσκευής **PDA, *7 (star 7)**
 - Δημιουργείται η θυγατρική εταιρία **FirstPerson Inc**
- Η δημιουργία των έξυπνων συσκευών αποτυγχάνει και η ομάδα (μαζί με τον **Eric Schmidt**) επικεντρώνεται στην εφαρμογή της πλατφόρμας στο **Internet**.
 - Ο Naughton φτιάχνει τον **WebRunner browser** (μετα **HotJava**)
 - Η γλώσσα μετονομάζεται σε **Java** και το ενδιαφέρον επικεντρώνεται σε εφαρμογές που τρέχουν μέσα στον browser.
- Ο **Marc Andersen** ανακοινώνει ότι ο **Netscape browser** θα υποστηρίζει Java μικροεφαρμογές (applets)

Java

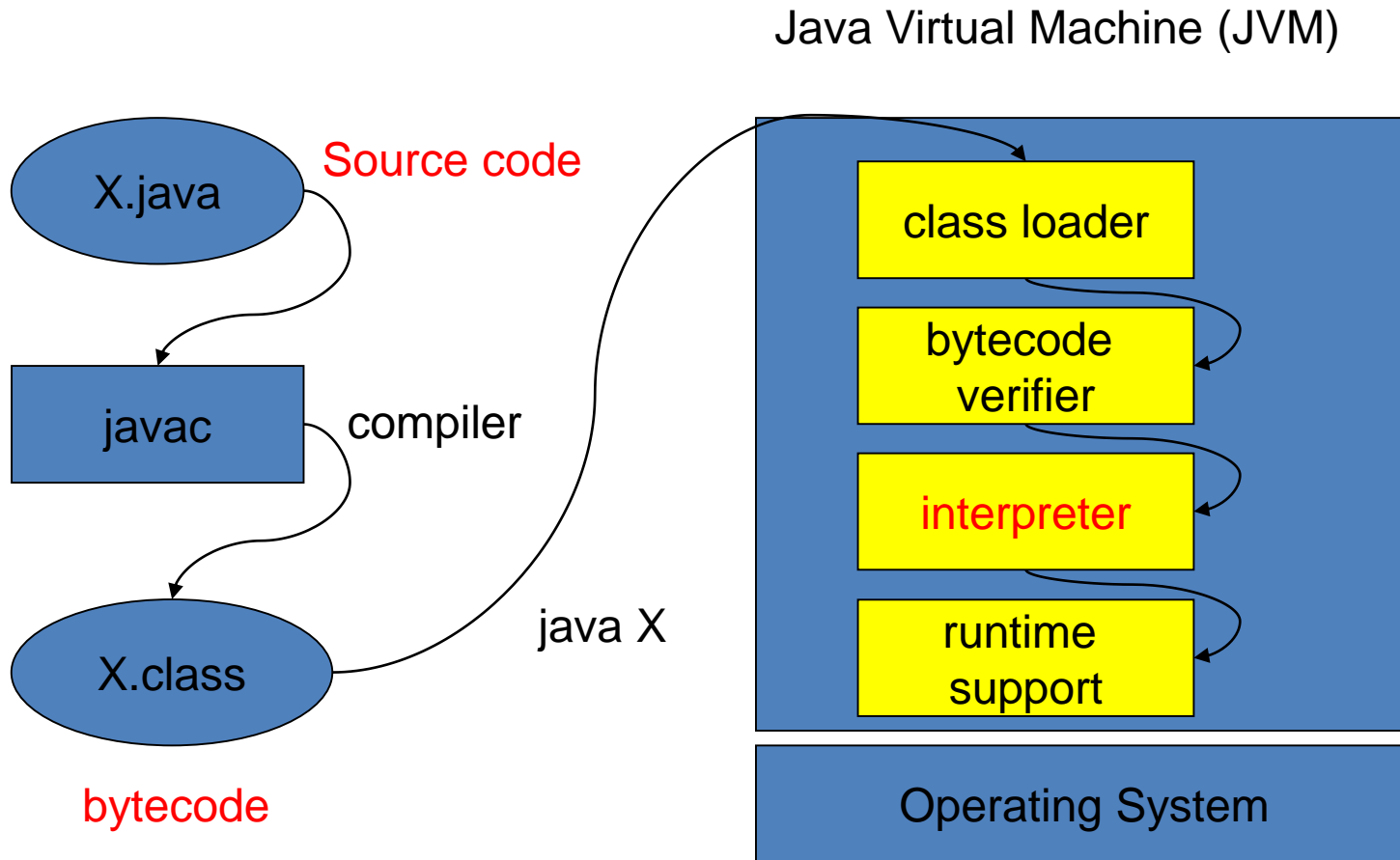
- Η Java είχε τους εξής στόχους:
 - "simple, object-oriented and familiar"
 - "robust and secure"
 - "architecture-neutral and portable"
 - "high performance"
 - "interpreted, threaded, and dynamic"

Java

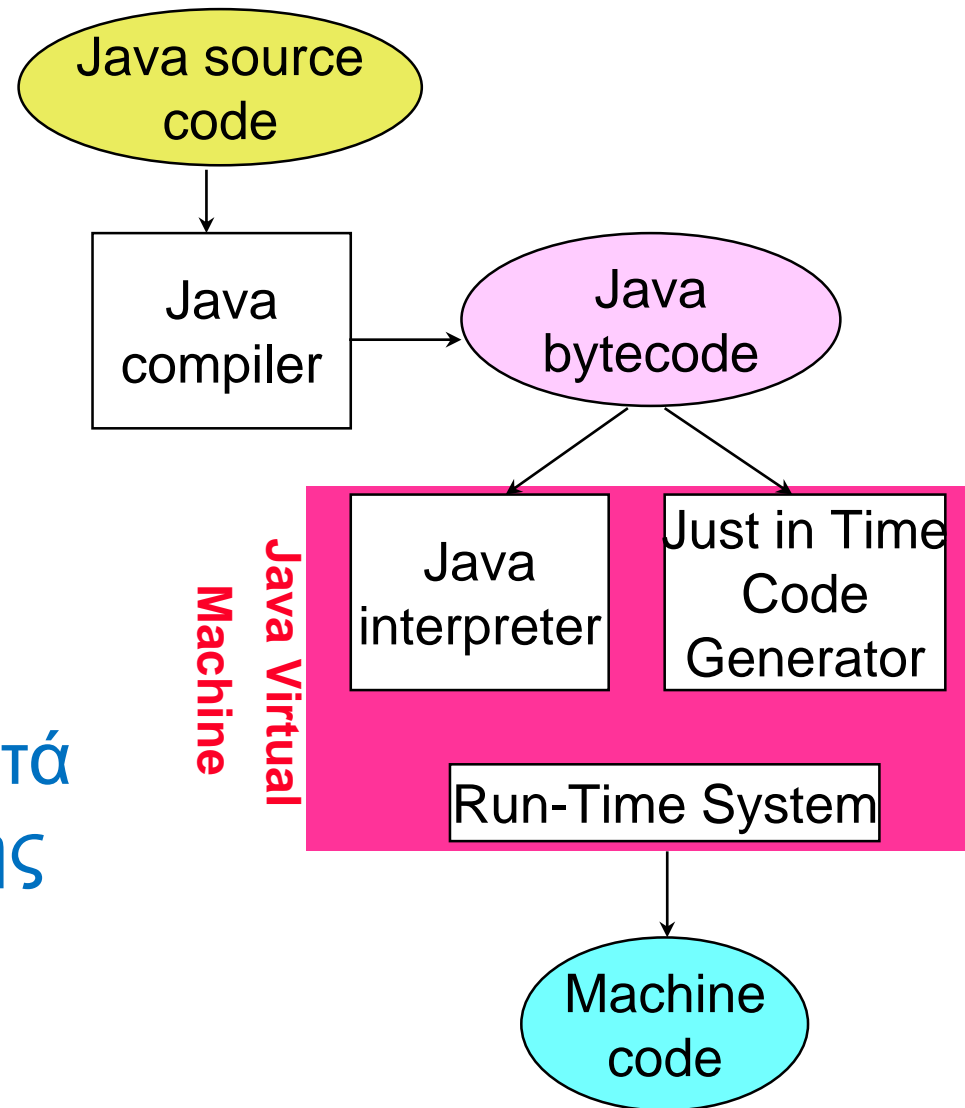
- Η Java είχε τους εξής στόχους:
 - "simple, object-oriented and familiar"
 - "robust and secure"
 - "architecture-neutral and portable"
 - "high performance"
 - "interpreted, threaded, and dynamic"

“architecture-neutral and portable”

- Το μεγαλύτερο πλεονέκτημα της Java είναι η **μεταφερισιμότητα (portability)**: ο κώδικας μπορεί να τρέξει πάνω σε οποιαδήποτε πλατφόρμα.
 - **Write-Once-Run-Anywhere** μοντέλο, σε αντίθεση με το σύνηθες **Write-Once-Compile-Anywhere** μοντέλο.
- Αυτό επιτυγχάνεται δημιουργώντας ένα **ενδιάμεσο κώδικα (bytecode)** ο οποίος μετά τρέχει πάνω σε μια **εικονική μηχανή (Java Virtual Machine)** η οποία το μεταφράζει σε **γλώσσα μηχανής**.
 - Οι προγραμματιστές πλέον γράφουν κώδικα για την εικονική μηχανή, η οποία δημιουργείται **για οποιαδήποτε πλατφόρμα**.



- **Just in Time (JIT) code generator (compiler)** βελτιώνει την απόδοση των Java Applications μεταφράζοντας (compiling) bytecode σε machine code **πριν ή κατά τη διάρκεια της εκτέλεσης**

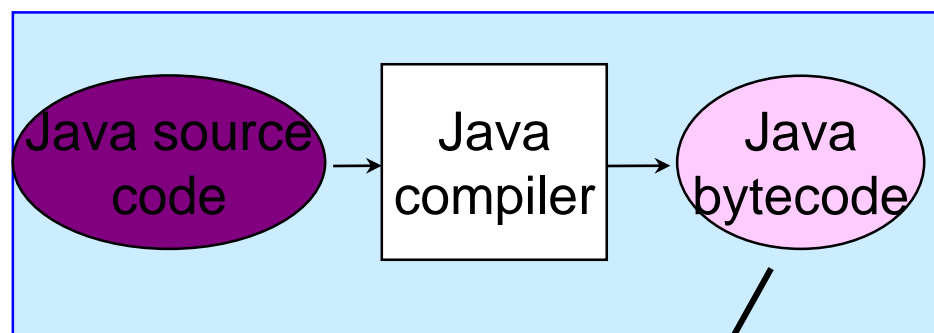


Java και το Internet

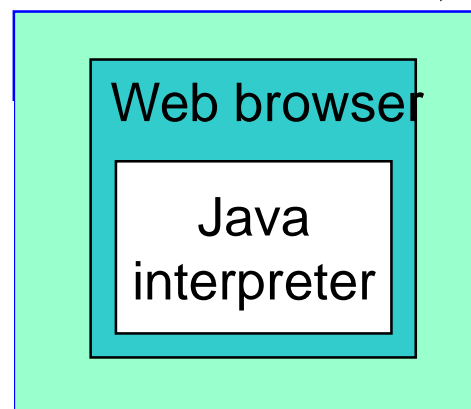
- Η προσέγγιση της Java είχε μεγάλη επιτυχία για **Web εφαρμογές**, όπου έχουμε ένα τεράστιο κατακευματισμένο **client-server** μοντέλο με πολλές διαφορετικές αρχιτεκτονικές
 - **Client-side programming**: Αντί να κάνει όλη τη δουλειά ο server για την δημιουργία της σελίδας κάποια από την επεξεργασία των δεδομένων γίνεται στη μηχανή του client.
 - **Web Applets**: κώδικας ο οποίος κατεβαίνει μαζί με τη Web σελίδα και τρέχει στη μηχανή του client. Είναι πολύ σημαντικό στην περίπτωση αυτή ο κώδικας να είναι portable.
 - **Server-side programming**: μία web σελίδα μπορεί να είναι το αποτέλεσμα ενός προγράμματος που συνδυάζει δυναμικά δεδομένα και είσοδο του χρήστη.
 - **Java Service Pages (JSPs)**: Η λύση της Java. Γίνεται compiled σε **servlets** και τρέχει στη μεριά του server.

Java Applets

- Το Web Browser software περιλαμβάνει ένα **JVM**
 - ◆ **Φορτώνει** τον java byte code από τον remote υπολογιστή
 - ◆ **Τρέχει** τοπικά το Java πρόγραμμα μέσα στο παράθυρο του Browser



Remote computer



Local computer



"simple, object-oriented and familiar"

- **Familiar:** Η Java είχε ως έμπνευση της την C++, και δανείζεται αρκετά από τα χαρακτηριστικά της.
- **Object-oriented:** Η Java είναι «**πιο αντικειμενοστραφής**» από την C++ η οποία προσπαθεί να μείνει συμβατή με την C
 - Στην Java **τα πάντα** είναι **αντικείμενα**
- **Simple:** Η Java δίνει λιγότερο έλεγχο στο χρήστη, αλλά κάνει τη ζωή του πιο εύκολη. Η **διαχείριση της μνήμης** γίνεται **αυτόματα**.
 - Η γλώσσα φροντίζει να κάνει πιο γρήγορο και πιο σταθερό (robust) τον προγραμματισμό παρότι αυτό μπορεί να έχει αποτέλεσμα τα προγράμματα να γίνονται **πιο αργά**.

HELLO WORLD

Το πρώτο μας πρόγραμμα σε Java

Java Installation

- Για να μπορείτε να μεταγλωττίσετε και να τρέξετε Java προγράμματα στον υπολογιστή σας θα πρέπει να **εγκαταστήσετε** την Java.
 - Θα κάνετε download and install από τη σελίδα της Oracle.
 - Ψάξετε “**download java**” ή “**install java**” για οδηγίες
 - <https://www.java.com/en/download/>
 - https://java.com/en/download/help/windows_manual_download.xml
- Υπάρχει περίπτωση μετά την εγκατάσταση να πρέπει να προσθέσετε το **path στο directory** στο οποίο εγκαταστάθηκε η Java στο **Path environmental variable**
 - Συνήθως αυτό γίνεται αυτόματα.

Δομή ενός απλού Java προγράμματος

- Το **όνομα** του αρχείου που κρατάει το πρόγραμμα είναι **X.java** (όπου **X** το όνομα του προγράμματος)
 - Στο παράδειγμα μας ονομάζουμε το πρόγραμμα μας: **HelloWorld.java**
- Μέσα στο πρόγραμμα μας πρέπει να έχουμε μια **κλάση** με το όνομα **X**.
 - **class X** (**class HelloWorld** στο παράδειγμα μας)
- Η κλάση **X** θα πρέπει να περιέχει μια **μέθοδο main** η οποία είναι το **σημείο εκκίνησης** του προγράμματος μας
 - **public static void main(String[] args)**

File HelloWorld.java

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Το **όνομα του .java αρχείου** και το **όνομα της κλάσης** (που περιέχει την μέθοδο main) θα πρέπει να είναι τα **ίδια** (αν η κλάση έχει οριστεί public)

Μεταγλώττιση – Compiling

Η μεταγλώττιση γίνεται με την εντολή **javac**

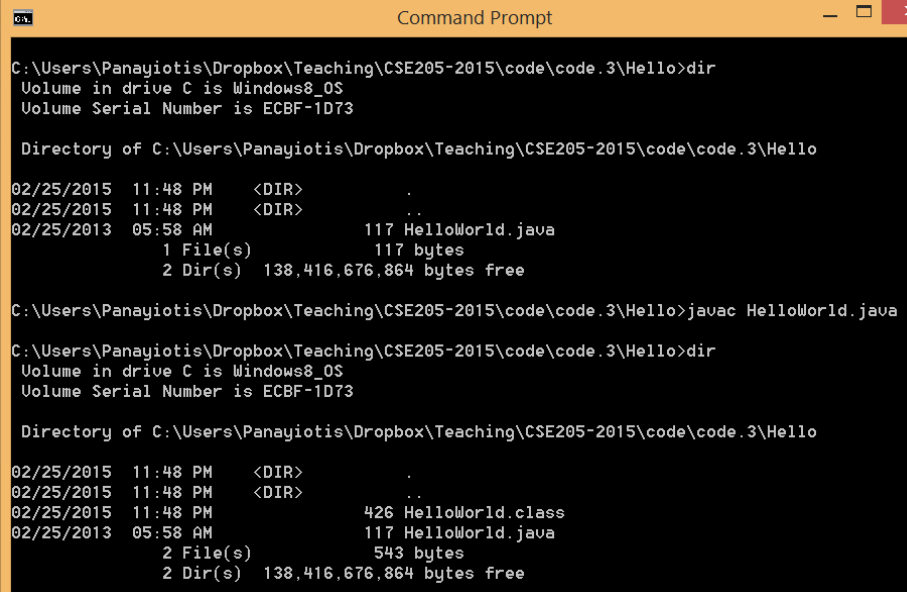
- `javac <.java αρχείο>`

Π.χ.

```
➤ javac HelloWorld.java
```

Το αποτέλεσμα είναι η δημιουργία ενός **.class** αρχείου που περιέχει τον ενδιαμέσο κώδικα (bytecode)

Το αρχείο **HelloWorld.class** στο παράδειγμα μας



```
Command Prompt
C:\Users\Panayiotis\Dropbox\Teaching\CSE205-2015\code\code.3\Hello>dir
Volume in drive C is Windows8_OS
Volume Serial Number is ECBF-1D73

Directory of C:\Users\Panayiotis\Dropbox\Teaching\CSE205-2015\code\code.3\Hello
02/25/2015  11:48 PM  <DIR>          .
02/25/2015  11:48 PM  <DIR>          ..
02/25/2013  05:58 AM              117 HelloWorld.java
                1 File(s)          117 bytes
                2 Dir(s)  138,416,676,864 bytes free

C:\Users\Panayiotis\Dropbox\Teaching\CSE205-2015\code\code.3\Hello>javac HelloWorld.java

C:\Users\Panayiotis\Dropbox\Teaching\CSE205-2015\code\code.3\Hello>dir
Volume in drive C is Windows8_OS
Volume Serial Number is ECBF-1D73

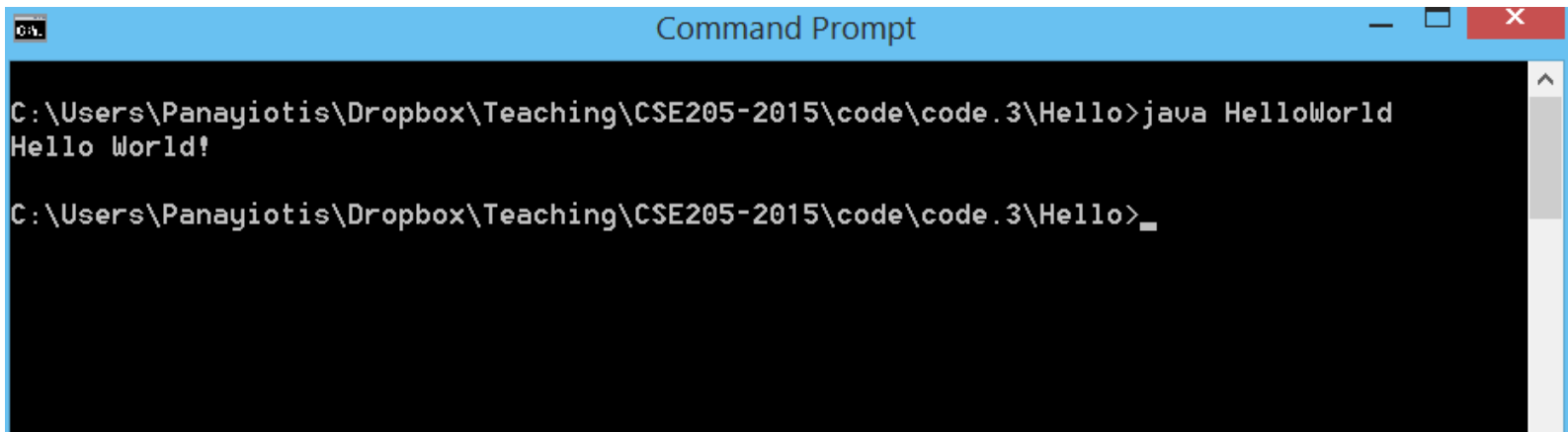
Directory of C:\Users\Panayiotis\Dropbox\Teaching\CSE205-2015\code\code.3\Hello
02/25/2015  11:48 PM  <DIR>          .
02/25/2015  11:48 PM  <DIR>          ..
02/25/2015  11:48 PM              426 HelloWorld.class
02/25/2013  05:58 AM              117 HelloWorld.java
                2 File(s)          543 bytes
                2 Dir(s)  138,416,676,864 bytes free
```

Εκτέλεση - Running

- Η εκτέλεση του κώδικα γίνεται με την εντολή **java**
 - `java <όνομα αρχείου χωρίς επίθεμα>`

➤ `java HelloWorld`

Χωρίς κανένα επίθεμα!



```
Command Prompt
C:\Users\Panayiotis\Dropbox\Teaching\CSE205-2015\code\code.3\Hello>java HelloWorld
Hello World!
C:\Users\Panayiotis\Dropbox\Teaching\CSE205-2015\code\code.3\Hello>_
```

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Λέξεις σε κόκκινο: δεσμευμένες λέξεις

Ορίζει την
κλάση

Όνομα της κλάσης

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

```
class HelloWorld
```

```
{
```

```
    public static void main(String args[])
```

```
    {
```

```
        // print message
```

```
        System.out.println("Hello world!");
```

```
    }
```

```
}
```

Τα άγκιστρα { ... } ορίζουν ένα **λογικό block** του κώδικα

- Αυτό μπορεί να είναι **μία κλάση**, **μία συνάρτηση**, **ένα if statement**
- Οι μεταβλητές που ορίζουμε μέσα σε ένα λογικό block, έχουν **εμβέλεια** μέσα στο block
- Αντίστοιχο των tabs στην Python, εδώ δεν χρειάζονται αλλά είναι καλό να τα βάζουμε για να διαβάζεται ο κώδικας πιο εύκολα.

Ορισμός της συνάρτησης main

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```


Ορισμός της συνάρτησης main

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

public, static: θα τα εξηγήσουμε αργότερα

Ορισμός της συνάρτησης main

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Το τι επιστρέφει η μέθοδος

void: Η μέθοδος δεν επιστρέφει τίποτα.

Ορισμός της συνάρτησης main

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Το όνομα της μεθόδου

- **main**: ειδική περίπτωση που σηματοδοτεί το σημείο εκκίνησης του προγράμματος.

Ορισμός της συνάρτησης main

```
class HelloWorld
{
    public static void main (String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Ορίσματα της μεθόδου

- Ένας **πίνακας** από **Strings** που αντιστοιχούν στις **παραμέτρους** με τις οποίες τρέχουμε το πρόγραμμα.

Η κλάση String

```
class HelloWorld
{
    public static void main (String args [])
    {
        // print message
        System.out.println ("Hello world!");
    }
}
```

- **String**: κλάση που χειρίζεται τα **αλφαριθμητικά**.
- Στη Java χρειάζεται να ορίσουμε τον **τύπο** της κάθε μεταβλητής
- **Strongly typed language**

Σχόλια!

```
/**  
 * A class that prints a message "hello world"  
 **/
```

```
class HelloWorld
```

```
{
```

```
    public static void main(String args[])
```

```
    {
```

```
        // print message
```

```
        System.out.println("Hello world!");
```

```
    }
```

```
}
```

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Κάθε εντολή στη Java πρέπει να τερματίζει με το ;

```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Αντικείμενο **System.out**
Ορίζει το ρεύμα εξόδου

Μέθοδος println:
Τυπώνει το String αντικείμενο που
δίνεται ως όρισμα και αλλάζει γραμμή


```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Το "Hello World" είναι ένα αντικείμενο της κλάσης String

Programming Style

Το όνομα της κλάσης ξεκινάει με κεφαλαίο και χρησιμοποιούμε την **CamelCase** σύμβαση

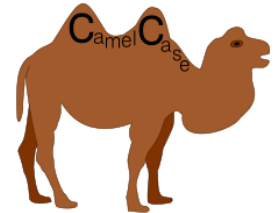
```
class HelloWorld
{
    public static void main(String args[])
    {
        // print message
        System.out.println("Hello world!");
    }
}
```

Στοίχιση του κώδικα:

- Οι εντολές μέσα σε ένα block του κώδικα ξεκινάνε ένα tab πιο μπροστά από το προηγούμενο.
- Όλες οι εντολές σε ένα block είναι στοιχισμένες
- Τα άγκιστρα είναι στοιχισμένα με την εντολή που ορίζει το block

Programming Style: Ονόματα

- Τα ονόματα των **κλάσεων** ξεκινάνε με κεφαλαίο, τα ονόματα των **πεδίων**, **μεθόδων** και **αντικειμένων** με μικρό.
 - Π.χ., **HelloWorld**, **position**, **print**
- Κάποιες συμβάσεις ξεκινούν τα ονόματα πεδίων με ‘**_**’ για να ξεχωρίζουν από τις μεθόδους
 - Π.χ., **_position**
- Χρησιμοποιούμε **ολόκληρες λέξεις** (και συνδυασμούς τους) για τα ονόματα
 - Δεν πειράζει αν βγαίνουν μεγάλα ονόματα
- Χρησιμοποιούμε το **CamelCase Style**
 - Όταν για ένα όνομα έχουμε πάνω από μία λέξη, τις συνενώνουμε και στο σημείο συνένωσης κάνουμε το πρώτο γράμμα της λέξης κεφαλαίο
 - **printName** όχι **print_name**
- Χρησιμοποιούμε **κεφαλαία** και ‘**_**’ για τις **σταθερές**.
 - Π.χ., **PI_NUMBER**



Παράδειγμα 2

- Φτιάξτε ένα πρόγραμμα που τυπώνει το λόγο δύο ακεραίων.

Division.java

Ισοδύναμο με το `String args[]`

```
class Division
{
    public static void main(String[] args)
    {
        int enumerator = 32;
        int denominator = 10;
        double division;
        division = enumerator / (double) denominator;
        System.out.println("Result = " + division);
    }
}
```

Division.java

```
class Division
{
    public static void main(String args[])
    {
        int enumerator = 32;
        int denominator = 10;
        double division;
        division = enumerator / (double) denominator;
        System.out.println("Result = " + division);
    }
}
```

Δήλωση δύο ακεραίων μεταβλητών και μιας πραγματικής μεταβλητής

- Ορισμός μεταβλητών
- Η Java είναι **strongly typed** γλώσσα: κάθε μεταβλητή θα πρέπει να έχει ένα **τύπο**.
- Οι τύποι **int** και **double** είναι **πρωταρχικοί (βασικοί) τύποι (primitive types)**
- Εκτός από τους βασικούς τύπους, όλοι οι άλλοι **τύποι** είναι **κλάσεις**

Ορισμός/Δήλωση μεταβλητών

- Ορισμός μεταβλητής

```
<τυπος> <όνομα μεταβλητής> [ = τιμή ] ;
```

- Ο ορισμός της μεταβλητής γίνεται **μόνο μία φορά**, πριν ή όταν θα την χρησιμοποιήσουμε για πρώτη φορά.
- Ο τύπος της μεταβλητής είναι είτε ένας πρωταρχικός τύπος, είτε μια υπάρχουσα ή νέα κλάση

- Παραδείγματα

```
int enumerator = 32;
```

Ορισμός μεταβλητής με αρχικοποίηση

```
int denominator = 10;
```

```
String myString;
```

Ορισμός μεταβλητής χωρίς αρχικοποίηση. Η Java θα δώσει default τιμές ("", false)

```
boolean b;
```

```
double division = enumerator / (double) denominator;
```

Ορισμός μεταβλητής με αρχικοποίηση με ανάθεση

Πρωταρχικοί τύποι

Όνομα τύπου	Τιμή	Μνήμη
boolean	true/false	1 byte
char	Χαρακτήρας (Unicode)	2 bytes
byte	Ακέραιος	1 byte
short	Ακέραιος	2 bytes
int	Ακέραιος	4 bytes
long	Ακέραιος	8 bytes
float	Πραγματικός	4 bytes
double	Πραγματικός	8 bytes

Όταν ορίζουμε μια μεταβλητή **δεσμεύεται** ο αντίστοιχος χώρος στη **μνήμη**. Το **όνομα της μεταβλητής** αντιστοιχίζεται με αυτό το χώρο στη **μνήμη**.

Πρωταρχικοί τύποι

Όνομα τύπου	Τιμή	Μνήμη
boolean	true/false	1 byte
char	Χαρακτήρας (Unicode)	2 bytes
byte	Ακέραιος	1 byte
short	Ακέραιος	2 bytes
int	Ακέραιος	4 bytes
long	Ακέραιος	8 bytes
float	Πραγματικός	4 bytes
double	Πραγματικός	8 bytes

Όταν ορίζουμε μια μεταβλητή **δεσμεύεται** ο αντίστοιχος χώρος στη **μνήμη**. Το **όνομα της μεταβλητής** αντιστοιχίζεται με αυτό το χώρο στη **μνήμη**.

Division.java

```
class Division
{
    public static void main(String args[])
    {
        int enumerator = 32;
        int denominator = 10;
        double division;
        division = enumerator / (double) denominator;
        System.out.println("Result = " + division);
    }
}
```

Ανάθεση: αποτίμηση της τιμής της έκφρασης στο δεξιό μέλος του “=” και μετά ανάθεση της τιμής στην μεταβλητή στο αριστερό μέλος

Το αριστερό μέλος είναι **πάντα** μεταβλητή

Division.java

```
class Division
{
    public static void main(String args[])
    {
        int enumerator = 32;
        int denominator = 10;
        double division;
        division = enumerator / (double) denominator;
        System.out.println("Result = " + division);
    }
}
```

Μετατροπή τύπου (type casting): `(double) denominator` μετατρέπει την τιμή της μεταβλητής `denominator` σε `double`.

Αν δεν γίνει η μετατροπή, η διαίρεση μεταξύ ακεραίων μας δίνει **πάντα** ακέραιο.

Αναθέσεις

- Στην ανάθεση κατά κανόνα, η τιμή του δεξιού μέρους θα πρέπει να είναι **ίδιου τύπου** με την μεταβλητή του αριστερού μέρους.
- Υπάρχουν εξαιρέσεις όταν υπάρχει **συμβατότητα** μεταξύ τύπων
- **byte** → **short** → **int** → **long** → **float** → **double**
 - Μια τιμή τύπου **T** μπορούμε να την αναθέσουμε σε μια μεταβλητή τύπου που εμφανίζεται **δεξιά του T**.
- (Σε αντίθεση με την C) ο τύπος **boolean** δεν είναι συμβατός με τους ακέραιους.

Division.java

```
class Division
{
    public static void main(String args[])
    {
        int enumerator = 32;
        int denominator = 10;
        double division;
        division = enumerator / (double) denominator;
        System.out.println("Result = " + division);
    }
}
```

Ο τελεστής “+” μεταξύ αντικείμενων της κλάσης String **συνενώνει** (concatenates) τα δύο String.

Μεταξύ ενός String και ενός βασικού τύπου, ο βασικός τύπος **μετατρέπεται** σε String και γίνεται η συνένωση

Η μνήμη του υπολογιστή

- Η **κύρια μνήμη** (main memory) του υπολογιστή κρατάει τα **δεδομένα** (και τις εντολές) για την εκτέλεση των προγραμμάτων.
 - Η μνήμη είναι προσωρινή, τα δεδομένα χάνονται όταν ολοκληρωθεί το πρόγραμμα.
- Η μνήμη είναι χωρισμένη σε **bytes** (8 bits)
 - Ο χώρος που χρειάζεται για ένα **χαρακτήρα ASCII**.
- Το κάθε byte έχει μια **διεύθυνση**, με την οποία μπορούμε να προσπελάσουμε τη συγκεκριμένη θέση μνήμης
 - **Random Access Memory (RAM)**
 - Σε 32-bit συστήματα μια διεύθυνση είναι 32 bits, σε 64-bit συστήματα μια διεύθυνση είναι 64 bits.

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	'a'
0001	'b'
0010	'c'
0011	'd'
0100	'e'
0101	'f'
0110	'g'
0111	'h'

Αποθήκευση μεταβλητών

- Η **κύρια μνήμη** (main memory) του υπολογιστή κρατάει τις **μεταβλητές** ενός προγράμματος
- Μια μεταβλητή μπορεί να απαιτεί χώρο περισσότερο από 1 byte.
 - Π.χ., οι μεταβλητές τύπου double χρειάζονται 8 bytes.
 - Η μεταβλητή τότε αποθηκεύεται σε συνεχόμενα bytes στη μνήμη.
- Η **θέση μνήμης** (διεύθυνση) της μεταβλητής θεωρείται το **πρώτο byte** από το οποίο ξεκινάει η αποθήκευση του της μεταβλητής.
 - Στο παράδειγμα μας η μεταβλητή βρίσκεται στη θέση 0000
 - Αν ξέρουμε την αρχή και το μέγεθος της μεταβλητής μπορούμε να τη διαβάσουμε.
- Άρα μία **θέση μνήμης** αποτελείται από μία **διεύθυνση** και το **μέγεθος**.

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	8.5
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Αποθήκευση μεταβλητών πρωταρχικού τύπου

- Για τις μεταβλητές **πρωταρχικού** τύπου (char, int, double,...) ξέρουμε εκ των προτέρων το μέγεθος της μνήμης που χρειαζόμαστε.
- Όταν ο μεταγλωττιστής δει τη **δήλωση** μιας μεταβλητής πρωταρχικού τύπου **δεσμεύει** μια θέση μνήμης αντίστοιχου μεγέθους
 - Η δήλωση μιας μεταβλητής ουσιαστικά **δίνει ένα όνομα** σε μία θέση μνήμης
 - Συχνά λέμε η **θέση μνήμης x** για τη μεταβλητή **x**.

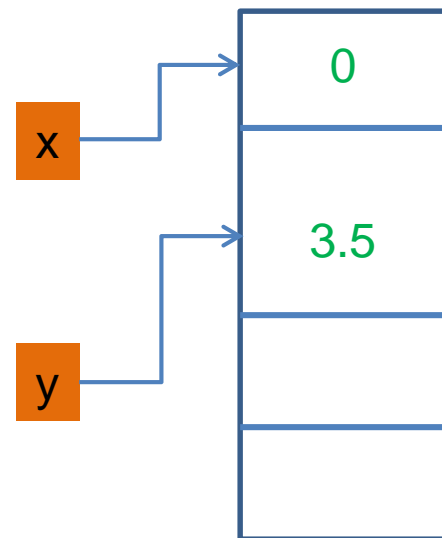
```
int x = 5;  
int y = 3;
```

	Διεύθυνση μνήμης	Περιεχόμενο μνήμης
x	0000	5
	0001	
	0010	
	0011	
y	0100	3
	0101	
	0110	
	0111	

Αποθήκευση μεταβλητών

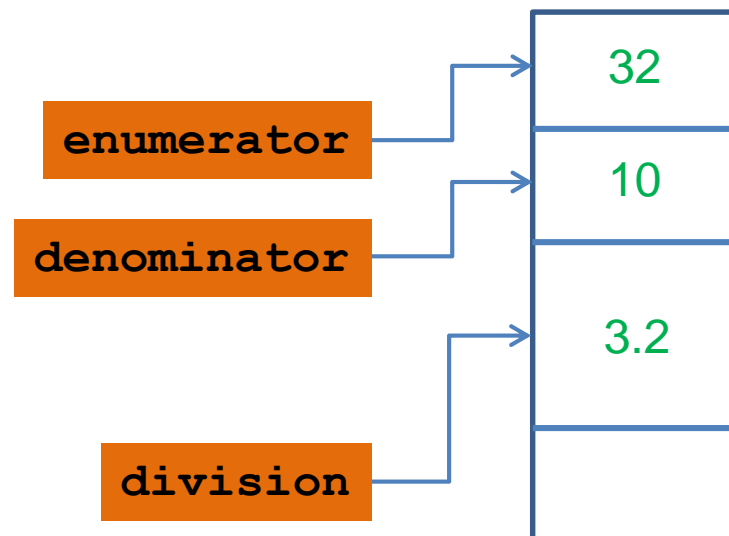
- Μπορούμε να σκεφτόμαστε την μνήμη του υπολογιστή σαν μια σειρά από «**κουτάκια**» διαφόρων μεγεθών στα οποία μπορούμε να αποθηκεύουμε δεδομένα
 - Το κάθε κουτάκι έχει **διεύθυνση** και **περιεχόμενα**
- Όταν **ορίζουμε** μια μεταβλητή πρωταρχικού τύπου (π.χ., **int x**) αυτό που γίνεται είναι ότι:
 - **Δεσμεύουμε** ένα κουτάκι κατάλληλου μεγέθους
 - 4 bytes για ένα int
 - Δίνουμε στο κουτάκι το **όνομα** της μεταβλητής
 - Το κουτάκι **x** αντί για το κουτάκι **0110**
 - Γι αυτό και η μεταβλητή **ορίζεται** μόνο μια φορά.
- Με την ανάθεση αλλάζουμε τα **περιεχόμενα** του κουτιού
- Αν δεν αρχικοποιήσουμε μια μεταβλητή η Java την αρχικοποιεί στο μηδέν (ή το αντίστοιχο του μηδέν για μη αριθμητικές μεταβλητές)

```
int x;  
double y = 3.5;
```



Division.java

```
class Division
{
    public static void main(String args[])
    {
        int enumerator = 32;
        int denominator = 10;
        double division;
        division =
            enumerator / (double) denominator;
        System.out.println(
            "Result = " + division);
    }
}
```



Ανάθεση: Διαβάζουμε τα περιεχόμενα των μεταβλητών **enumerator** και **denominator** κάνουμε τον υπολογισμό και αλλάζουμε τα περιεχόμενα της μεταβλητής **division** αποθηκεύοντας το αποτέλεσμα της διαίρεσης.

Strings

- Η κλάση `String` είναι **προκαθορισμένη κλάση** της Java που μας επιτρέπει να χειριζόμαστε αλφαριθμητικά.
 - Είναι διαφορετική από τους πρωταρχικούς τύπους αλλά η Java μας δίνει περισσότερη ευελιξία
- Ο τελεστής “+” μας επιτρέπει την **συνένωση**
- Υπάρχουν πολλές χρήσιμες **μέθοδοι** της κλάσης `String`. Η πιο χρήσιμη αυτή τη στιγμή είναι:
 - `equals(String x)`: ελέγχει για ισότητα του αντικειμένου που κάλεσε την μέθοδο και του ορίσματος `x`.
- Άλλες μέθοδοι:
 - `length()`: μήκος του `String`
 - `trim()`: αφαιρεί κενά στην αρχή και το τέλος του `string`.
 - `split(char delim)`: σπάει το `string` σε πίνακα από `strings` με βάση το χαρακτήρα `delim`.
 - Επίσης, μέθοδοι για να βρεθεί ένα υπο-`string` μέσα σε ένα `string`, κλπ.

Escape sequences

- Για να τυπώσουμε κάποιους ειδικούς χαρακτήρες (π.χ., τον χαρακτήρα “) χρησιμοποιούμε τον χαρακτήρα \ και μετά τον χαρακτήρα που θέλουμε να τυπώσουμε
 - Π.χ., ακολουθία \”
- Αυτό ισχύει γενικά για ειδικούς χαρακτήρες.

• \b	Backspace
• \t	Tab
• \n	New line
• \f	Form feed
• \r	Return (ENTER)
• \”	Double quote
• \’	Single quote
• \\	Backslash
• \ddd	Octal code
• \uxxxx	Hex-decimal code

Παράδειγμα

```
class StringTest
{
    public static void main(String args[])
    {
        String s = "hello world";
        String h = "hello";
        String w = "world";
        System.out.println(s);
        System.out.println(h+"\t"+w);
        System.out.println(s.length());
        System.out.println("hello".length());
    }
}
```

Ρεύματα εισόδου/εξόδου

- Τι είναι ένα ρεύμα? Μια **αφαίρεση** που αναπαριστά μια **πηγή** (για την **είσοδο**), ή ένα **προορισμό** (για την **έξοδο**) **χαρακτήρων**
 - Αυτό μπορεί να είναι ένα αρχείο, το πληκτρολόγιο, η οθόνη.
 - Όταν δημιουργούμε το ρεύμα το **συνδέουμε** με την ανάλογη **πηγή**, ή **προορισμό**.

Είσοδος & Έξοδος

- Τα βασικά ρεύματα εισόδου/εξόδου είναι έτοιμα **αντικείμενα** τα οποία ορίζονται σαν πεδία (**στατικά**) της κλάσης **System**
 - **System.out**
 - **System.in**
 - **System.err**
- Μέσω αυτών και άλλων βοηθητικών αντικειμένων γίνεται η είσοδος και έξοδος δεδομένων ενός προγράμματος.
- Μια εντολή εισόδου/εξόδου έχει αποτέλεσμα το **λειτουργικό** να **πάρει ή να στείλει** **χαρακτήρες** από/προς την αντίστοιχη **πηγή/προορισμό**.

Έξοδος

- Μπορούμε να καλέσουμε τις μεθόδους του `System.out`:
 - `println(String s)`: για να τυπώσουμε ένα αλφαριθμητικό `s` και τον χαρακτήρα `'\n'` (αλλαγή γραμμής)
 - `print(String s)`: τυπώνει το `s` αλλά δεν αλλάζει γραμμή
 - `printf`: Formatted output
 - `printf("%d",myInt);` // τυπώνει ένα ακέραιο
 - `printf("%f",myDouble);` // τυπώνει ένα πραγματικό
 - `printf("%.2f",myDouble);` // τυπώνει ένα πραγματικό με δύο δεκαδικά

Είσοδος

- Χρησιμοποιούμε την κλάση Scanner της Java
 - `import java.util.Scanner;`
- Αρχικοποιείται με το ρεύμα εισόδου:
 - `Scanner in = new Scanner(System.in);`
- Μπορούμε να καλέσουμε μεθόδους της Scanner για να διαβάσουμε κάτι από την είσοδο
 - `nextLine()`: διαβάζει **μέχρι** να βρει τον χαρακτήρα `'\n'`. **Επιστρέφει** ένα **String**
 - `next()`: διαβάζει το επόμενο **String** μέχρι να βρει **λευκό** χαρακτήρα. **Επιστρέφει** ένα **String**
 - `nextInt()`: διαβάζει τον επόμενο **int**. **Επιστρέφει** ένα **Integer**
 - `nextDouble()`: διαβάζει τον επόμενο **double**. **Επιστρέφει** ένα **Double**
 - `nextBoolean()`: διαβάζει τον επόμενο **boolean**. **Επιστρέφει** ένα **Boolean**

Παράδειγμα

Με την εντολή αυτή φέρνουμε την κλάση Scanner μέσα στο πρόγραμμά μας ώστε να μπορούμε να φτιάξουμε αντικείμενα τύπου Scanner

```
import java.util.Scanner;
```

```
class TestIO
```

```
{
```

```
    public static void main(String args[])
```

```
    {
```

```
        System.out.println("Say something:");
```

```
        Scanner input = new Scanner(System.in);
```

```
        String line = input.nextLine();
```

```
        System.out.println(line);
```

```
    }
```

```
}
```

Ορισμός μεταβλητής String

new: δημιουργεί ένα αντικείμενο τύπου **Scanner** (μία **μεταβλητή**)

με το οποίο μπορούμε πλέον να διαβάζουμε από την είσοδο.

- Το αντικείμενο αυτό αναπαριστά το **πληκτρολόγιο** στο πρόγραμμά μας. Ένα αντικείμενο φτάνει για να διαβάσουμε πολλαπλές τιμές.

Παράδειγμα

```
import java.util.Scanner;

class TestIO2
{
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        double d= input.nextDouble();
        System.out.println("division by 4 = " + d/4);
        System.out.println("1+ (division by 4) = " +1+d/4);
        System.out.printf("1+ (division of %.2f by 4) = %.2f",d, 1+d/4);
    }
}
```

Το + λειτουργεί ως **concatenation** τελεστής μεταξύ Strings, άρα μετατρέπει τους αριθμούς σε Strings

Τι θα τυπώσει αυτό το πρόγραμμα?

Λογικές εκφράσεις και Λογικοί τελεστές

- **Λογικές σταθερές/τιμές:**
 - **true**: αληθές
 - **false**: ψευδές
- **Λογική έκφραση:** μια έκφραση που αποτιμάται σε μια λογική τιμή
- Παραδείγματα: Έλεγχος για βασικούς τύπους A,B:
 - Ισότητας: **(A == B)**
 - Ανισότητας: **(A != B)** ή **(!(A == B))**
 - Μεγαλύτερο/Μικρότερο ή ίσο: **(A <= B)** , **(A >= B)**
- Έλεγχος για μεταβλητές (**αντικείμενα**) οποιουδήποτε άλλου τύπου γίνεται με την μέθοδο **equals** (πρέπει να έχει οριστεί):
 - Ισότητας: **(A.equals(B))**
 - Ανισότητας: **(!A.equals(B))**
- **Λογικοί τελεστές** για λογικές εκφράσεις
 - Άρνηση: **!B**
 - ΚΑΙ: **(A && B)**
 - Ή: **(A || B)**

Έλεγχος ισότητας για Strings

- Αν έχουμε δύο μεταβλητές String για να ελέγξουμε αν έχουν την ίδια τιμή **πρέπει** να χρησιμοποιήσουμε την μέθοδο `equals`.
- Παράδειγμα:

```
| String firstString = "abc";  
| String secondString = "ABC";  
| boolean test1 = firstString.equals(secondString);  
| boolean test2 = firstString.equals("abc");
```

- Η παρακάτω εντολή **δεν είναι σωστή**

```
| boolean test3 = (firstString == secondString);
```

- Περνάει από τον compiler και σε κάποιες περιπτώσεις θα δουλέψει αλλά **δεν κάνει αυτό που θέλουμε**.

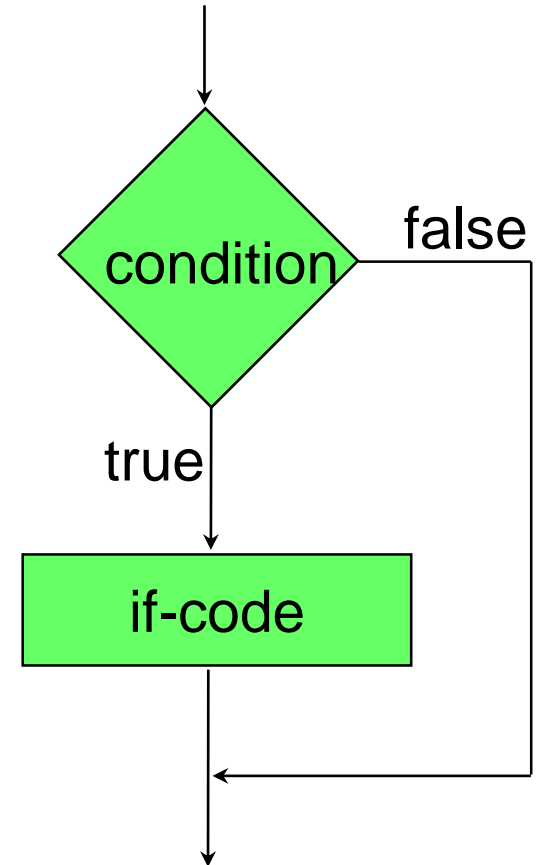
Βρόγχοι – Το if-then Statement

- Στην Java το **if-then statement** έχει το εξής συντακτικό

Η παρένθεση είναι απαραίτητη

```
if (condition)
{
    ...if-code block...
}
```

- Αν η **συνθήκη** είναι **αληθής** τότε εκτελείται το block κώδικα if-code
- Αν η **συνθήκη** είναι **ψευδής** τότε το κομμάτι αυτό προσπερνιέται και συνεχίζεται η εκτέλεση.




```
import java.util.Scanner;

class IfTest1b
{
    public static void main(String[] args)
    {
        Scanner reader = new Scanner(System.in);
        int inputInt = reader.nextInt();
        boolean inputIsPositive = (inputInt > 0)
        if (inputIsPositive) {
            System.out.println(inputInt +
                               " is positive");
        }
    }
}
```

Ακόμη και αν δεν το προσδιορίσουμε ελέγχει ισότητα

Programming Style: Λογικές μεταβλητές

- Συνηθίζεται όταν ορίζουμε **λογικές μεταβλητές** το **όνομα** τους να είναι αυτό που εκφράζει την περίπτωση που η μεταβλητή αποτιμάται **true**.

```
int x = 10;
boolean isPositive = (x > 0);
boolean isNegative = (x < 0);
boolean isNotPositive = !isPositive;
```

- Αυτό βολεύει για την εύκολη ανάγνωση του προγράμματος όταν χρησιμοποιούμε την μεταβλητή

```
if (isPositive) {
    System.out.println("Variable is positive");
}
```

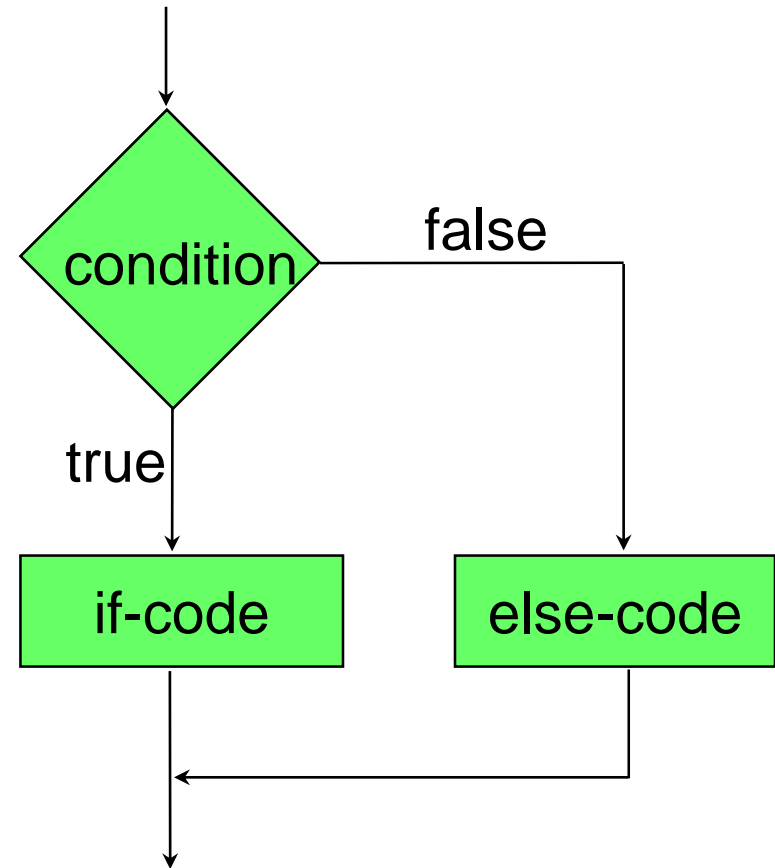
- Το ίδιο ισχύει και όταν αργότερα θα ορίζουμε **μεθόδους** που επιστρέφουν λογικές τιμές
 - Π.χ., για τα Strings υπάρχει η μέθοδος **equals** που γίνεται true όταν έχουμε ισότητα και η μέθοδος **isEmpty** που είναι true όταν έχουμε άδειο String.

Βρόγχοι – Το if-then-else Statement

- Στην Java το **if-then-else statement** έχει το εξής συντακτικό

```
| if (condition) {  
|     ...if-code block...  
| }else{  
|     ...else-code block...  
| }
```

- Αν η **συνθήκη** είναι **αληθής** τότε εκτελείται το block κώδικα if-code
- Αν η **συνθήκη** είναι **ψευδής** τότε εκτελείται το block κώδικα else-code.
- Ο κώδικας του if-code block ή του else-code block μπορεί να περιέχουν ένα άλλο (**φωλιασμένο (nested)**) if statement
- **Προσοχή:** ένα **else** clause ταιριάζεται με το **τελευταίο** ελεύθερο **if** ακόμη κι αν η στοίχιση του κώδικα υπονοεί διαφορετικά.



```
import java.util.Scanner;
```

```
class IfTest2
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Scanner reader = new Scanner(System.in);
```

```
        int inputInt = reader.nextInt();
```

```
        if (inputInt > 0) {
```

```
            System.out.println(inputInt +
```

```
                                " is positive");
```

```
        }else{
```

```
            System.out.println(inputInt +
```

```
                                " is not positive");
```

```
        }
```

```
    }
```

```
}
```

Έλεγχος πολλαπλών επιλογών

```
import java.util.Scanner;

class IfTest3
{
    public static void main(String[] args)
    {
        Scanner reader = new Scanner(System.in);
        int inputInt = reader.nextInt();
        if (inputInt > 0){
            System.out.println(inputInt +
                               " is positive");
        }else if (inputInt < 0){
            System.out.println(inputInt +
                               " is not positive");
        }else{
            System.out.println(inputInt + " is zero");
        }
    }
}
```

Προσοχή!

ΛΑΘΟΣ!

```
if( i == j )
    if ( j == k )
        System.out.print(
            "i equals k");
else
    System.out.print(
        "i is not equal to j");
```

Το else μοιάζει σαν να πηγαίνει με το μπλε else αλλά ταιριάζεται με το τελευταίο (πράσινο) if

ΣΩΣΤΟ!

```
if( i == j ){
    if ( j == k ){
        System.out.print(
            "i equals k");
    }
}
else {
    System.out.print(
        "i is not equal to j");
}
```

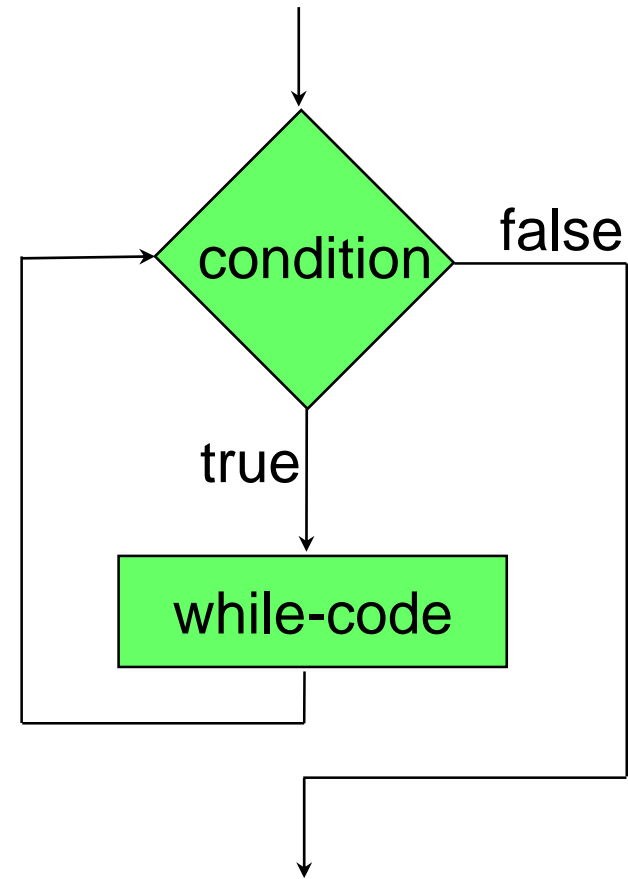
Πάντα να βάζετε `{ }` στο σώμα των if-then-else statements.
Πάντα να στοιχίζετε σωστά τον κώδικα.

Επαναλήψεις - While statement

- Στην Java το **while statement** έχει το εξής συντακτικό

```
while (condition)
{
    ...while-code block...
}
```

- Αν η **συνθήκη** είναι **αληθής** τότε εκτελείται το block κώδικα **while-code**
- Ο **while-code block** κώδικας υλοποιεί τις επαναλήψεις και **αλλάζει την συνθήκη**.
- Στο **τέλος του while-code block** η συνθήκη **αξιολογείται εκ νέου**
- Ο κώδικας επαναλαμβάνεται **μέχρι** η συνθήκη να γίνει **ψευδής**.



Παράδειγμα

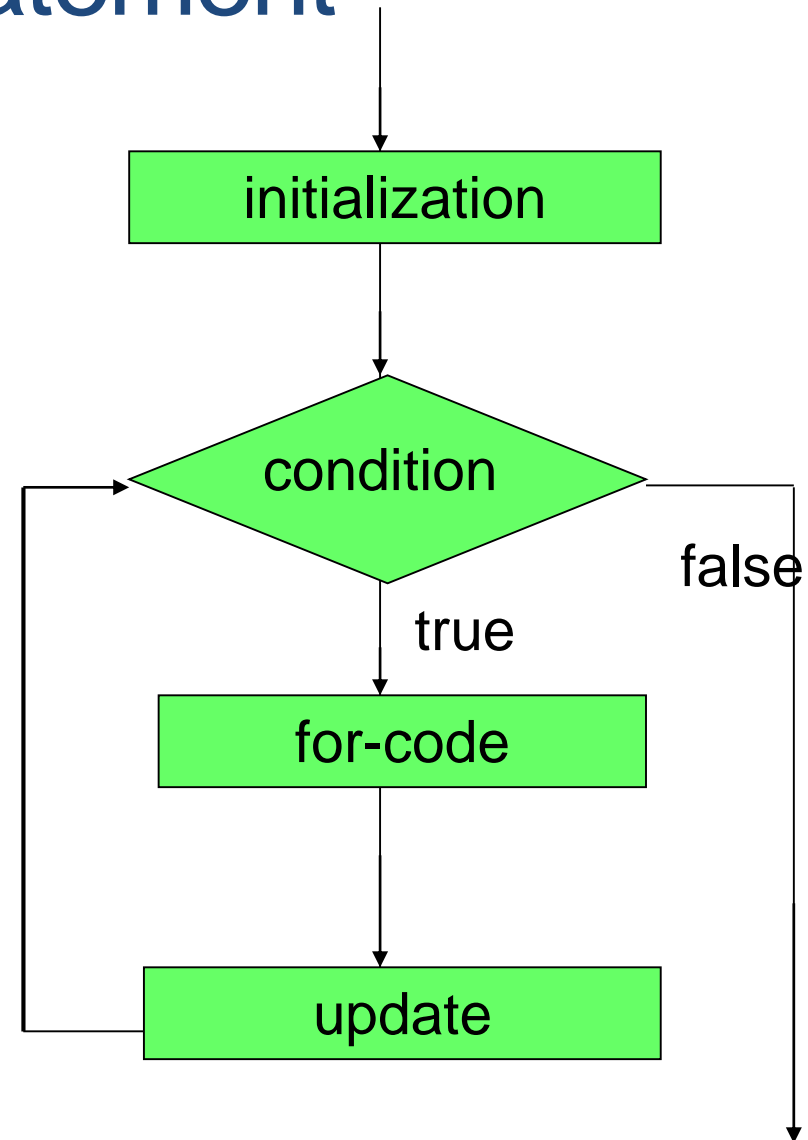
```
Scanner inputReader = new Scanner(System.in);  
String input = inputReader.next();  
  
while(input.equals("Yes"))  
{  
    System.out.println("Do you want to continue?");  
    input = inputReader.next();  
}
```


Επαναλήψεις – for statement

- Στην Java το **for statement** έχει το εξής ΣΥΝΤΑΚΤΙΚΟ

```
for (initialization;  
    condition;  
    update)  
{  
    ...for-code block...  
}
```

- Το όρισμα του for έχει 3 κομμάτια χωρισμένα με ;
 - Την **αρχικοποίηση (initialization section)**: εκτελείται πάντα μία μόνο φορά
 - Τη **λογική συνθήκη (condition)**: εκτιμάται πριν από κάθε επανάληψη.
 - Την **ενημέρωση (update expression)**: υπολογίζεται μετά το κυρίως σώμα της επανάληψης.
 - Ο κώδικας επαναλαμβάνεται **μέχρι** η συνθήκη να γίνει **ψευδής**.



Παράδειγμα

```
for (int i = 0; i < 10; i = i+1)
{
    System.out.println("i = " + i);
}
```

Ορισμός της μεταβλητής *i*

Ανάθεση: υπολογίζεται η τιμή του *i+1* και ανατίθεται στη μεταβλητή *i*.

- Ισοδύναμο με **while**

```
int i = 0;
while (i < 10)
{
    System.out.println("i = " + i);
    i = i+1;
}
```

Παράδειγμα

```
for(int i = 0; i < 10; i ++)  
{  
    System.out.println("i = " + i);  
    i = i+1;  
}
```

`i++` ισοδύναμο με το `i = i+1`

- Ισοδύναμο με **while**

```
int i = 0;  
while(i < 10)  
{  
    System.out.println("i = " + i);  
    i ++;  
}
```

Παράδειγμα

- Κάνετε πρόγραμμα που παίρνει σαν είσοδο ένα αριθμό και υλοποιεί μια αντίστροφη μέτρηση. Αν ο αριθμός είναι θετικός η αντίστροφη μέτρηση γίνεται προς τα κάτω μέχρι το μηδέν, αν είναι αρνητικός γίνεται προς τα πάνω μέχρι το μηδέν. Η διαδικασία επαναλαμβάνεται μέχρι ο χρήστης να δώσει την τιμή μηδέν.

```
import java.util.Scanner;

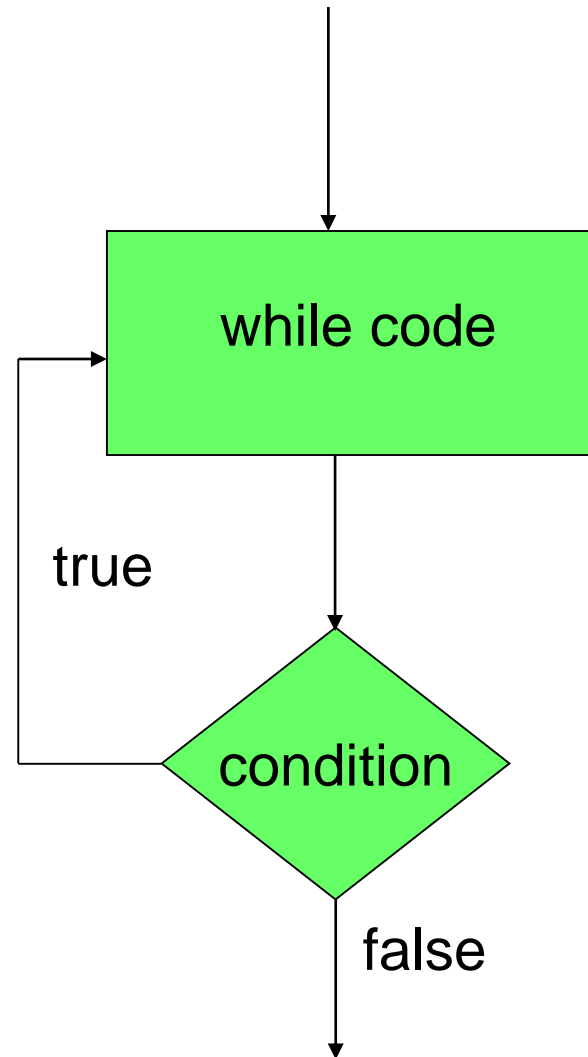
class Countdown
{
    public static void main(String[] args)
    {
        Scanner reader = new Scanner(System.in);
        int inputInt = reader.nextInt();
        while (inputInt != 0)
        {
            if (inputInt < 0 ){
                for (int i = inputInt; i < 0; i ++){
                    System.out.println("Counter = " + i);
                }
            } else if (inputInt > 0){
                for (int i = inputInt; i > 0; i --){
                    System.out.println("Counter = " + i);
                }
            }
            inputInt = reader.nextInt();
        }
    }
}
```

To Do-While statement

- Ένα **do while** statement έχει το εξής συντακτικό:

```
Initialize  
do  
{  
    ...while-code block...  
}while (condition)
```

- Το while code εκτελείται **τουλάχιστον μία φορά**; Μετά αν η συνθήκη είναι αληθής ο κώδικας εκτελείται ξανά.
- Οι μεταβλητές στο **condition** **δεν** μπορεί να είναι **τοπικές** μεταβλητές του while code



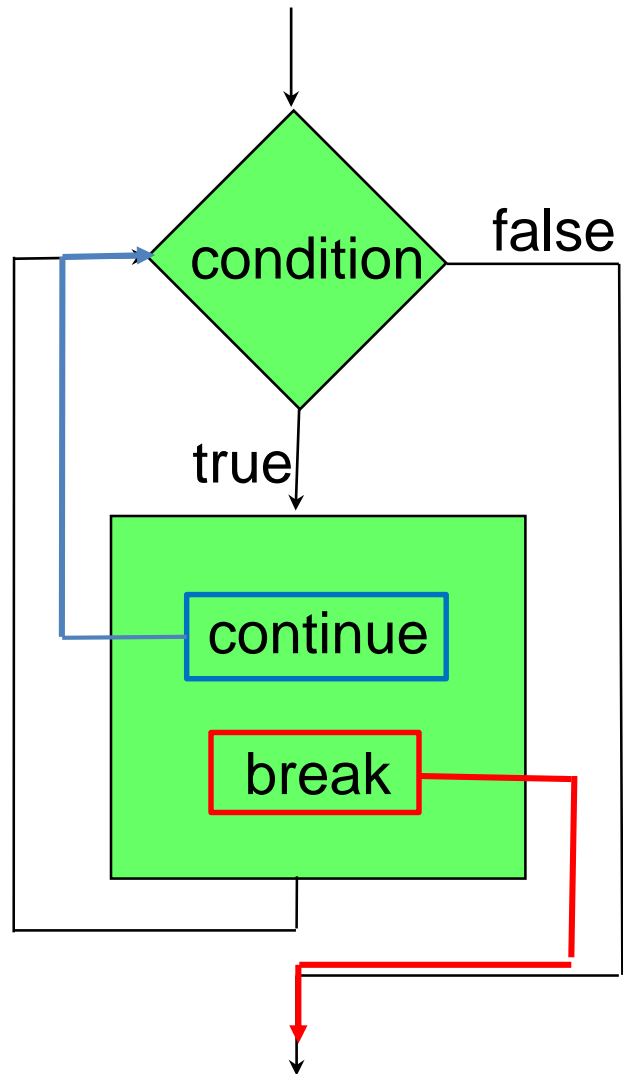
```
import java.util.Scanner;

class CountdownWithDo
{
    public static void main(String[] args)
    {
        Scanner reader = new Scanner(System.in);
        int inputInt;
        do
        {
            inputInt = reader.nextInt();
            if (inputInt < 0 ){
                for (int i = inputInt; i < 0; i ++){
                    System.out.println("Counter = " + i);
                }
            } else if (inputInt > 0){
                for (int i = inputInt; i > 0; i --){
                    System.out.println("Counter = " + i);
                }
            }
        } while (inputInt != 0)
    }
}
```

Οι εντολές `break` και `continue`

- **`continue`**: Επιστρέφει τη ροή του προγράμματος στον έλεγχο της συνθήκης σε ένα βρόγχο.
 - Βολικό για τον έλεγχο συνθηκών πριν ξεκινήσει η εκτέλεση του βρόγχου, ή για πρόορη επιστροφή στον έλεγχο της συνθήκης
- **`break`**: Μας βγάζει έξω από την εκτέλεση του βρόχου από οποιοδήποτε σημείο μέσα στον κώδικα.
 - Βολικό για να σταματάμε το βρόγχο όταν κάτι δεν πάει καλά.
- Κάποιοι θεωρούν οι εντολές αυτές χαλάνε το μοντέλο του δομημένου προγραμματισμού.

Οι εντολές break και continue



Παράδειγμα

```
while (...)  
{  
    if (everything is ok) {  
        < rest of code >  
    } // end of if  
} // end of while loop
```

```
while (... && !StopFlag)  
{  
    < some code >  
  
    if (I should stop) {  
        StopFlag = true;  
    } else {  
        < some more code >  
    }  
} // end of while loop
```

```
while (...)  
{  
    if (I don't like something) {  
        continue;  
    }  
  
    < rest of code >  
} // end of while loop
```

```
while (...)  
{  
    < some code >  
  
    if (I should stop) {  
        break;  
    }  
  
    < some code >  
} // end of while loop
```

```
import java.util.Scanner;

class CountdownWithContinue
{
    public static void main(String[] args)
    {
        Scanner reader = new Scanner(System.in);
        int inputInt = reader.nextInt();
        while (inputInt != 0)
        {
            if (inputInt%2 == 0){
                inputInt = reader.nextInt();
                continue;
            }
            if (inputInt < 0 ){
                for (int i = inputInt; i < 0; i ++){
                    System.out.println("Counter = " + i);
                }
            } else if (inputInt > 0){
                for (int i = inputInt; i > 0; i --){
                    System.out.println("Counter = " + i);
                }
            }
            inputInt = reader.nextInt();
        }
    }
}
```

Ο τελεστής % υπολογίζει
το υπόλοιπο διαίρεσης

Η αντίστροφη μέτρηση εκτελείται
μόνο για περιττούς αριθμούς

```
import java.util.Scanner;

class CountdownWithBreak
{
    public static void main(String[] args)
    {
        Scanner reader = new Scanner(System.in);
        do
        {
            int inputInt = reader.nextInt();
            if (inputInt == 0) {
                break;
            }
            if (inputInt < 0 ) {
                for (int i = inputInt; i < 0; i ++ )
                {
                    System.out.println("Counter = " + i);
                }
            } else if (inputInt > 0) {
                for (int i = inputInt; i > 0; i -- )
                {
                    System.out.println("Counter = " + i);
                }
            }
        } while (true)
    }
}
```

Η συνθήκη αυτή ορίζει ένα **ατέρμονο βρόγχο** (infinite loop). Πρέπει μέσα στο πρόγραμμα να έχουμε μια εντολή **break** (ή return που θα δούμε αργότερα) για να μην κολλήσει το πρόγραμμα μας. Αυτή η κατασκευή είναι βολική όταν έχουμε πολλαπλές συνθήκες εξόδου.

Εμβέλεια (scope) μεταβλητών

- Προσέξτε ότι η μεταβλητή `int i` πρέπει να οριστεί **σε** **κάθε** `for`, ενώ η `inputInt` πρέπει να οριστεί **έξω** από το `while-loop` αλλιώς ο compiler διαμαρτύρεται.
 - Προσπαθούμε να χρησιμοποιήσουμε μια μεταβλητή εκτός της **εμβέλειας** της
- Η κάθε μεταβλητή που ορίζουμε έχει **εμβέλεια (scope)** μέσα στο `block` το οποίο ορίζεται.
 - **Τοπική μεταβλητή** μέσα στο `block`.
- Μόλις **βγούμε** από το `block` η μεταβλητή χάνεται
 - Ο compiler δημιουργεί ένα χώρο στη μνήμη για το `block` το οποίο εκτελούμε, ο οποίος εξαφανίζεται όταν το `block` τελειώσει.
- Ένα `block` μπορεί να περιλαμβάνει κι άλλα **φωλιασμένα blocks**
 - Η μεταβλητή έχει **εμβέλεια** και μέσα στα **φωλιασμένα blocks**
 - **Δεν μπορούμε** να ορίσουμε μια άλλη **μεταβλητή με το ίδιο όνομα** σε ένα φωλιασμένο `block`

Παράδειγμα με το scope μεταβλητών

```
public static void main(String[] args)
{
    int y = 1;
    int x = 2;
    for (int i = 0; i < 3; i ++ )
    {
        y = i;
        double x = i+1;
        int z = x+y;
        System.out.println("i = " + i);
        System.out.println("y = " + y);
        System.out.println("z = " + z);
    }
    int z = 0;
    System.out.println("i = " + i);
    System.out.println("z = " + z);
    System.out.println("y = " + y);
    System.out.println("x = " + x);
}
```

Ο κώδικας έχει λάθη σε **τρία** σημεία

Δεν είναι λάθος
γιατί ο πρώτος
ορισμός είναι εκτός
εμβέλειας

```
public static void main(String[] args)
{
    ... ..
    {
        ... ..
        {
            int y = 1;
            ... ..
            {
                ... ..
            }
            ... ..
        }
        ... ..
    }
    ... ..
}
```

Η διαφορά του κόκκινου από το μπλε είναι ο χώρος **εκτός** της εμβέλειας του **y**

Έξω από το μπλε **δεν** μπορούμε να χρησιμοποιήσουμε τη μεταβλητή **y**, αλλά **μπορούμε** να **ορίσουμε** νέα μεταβλητή **y**

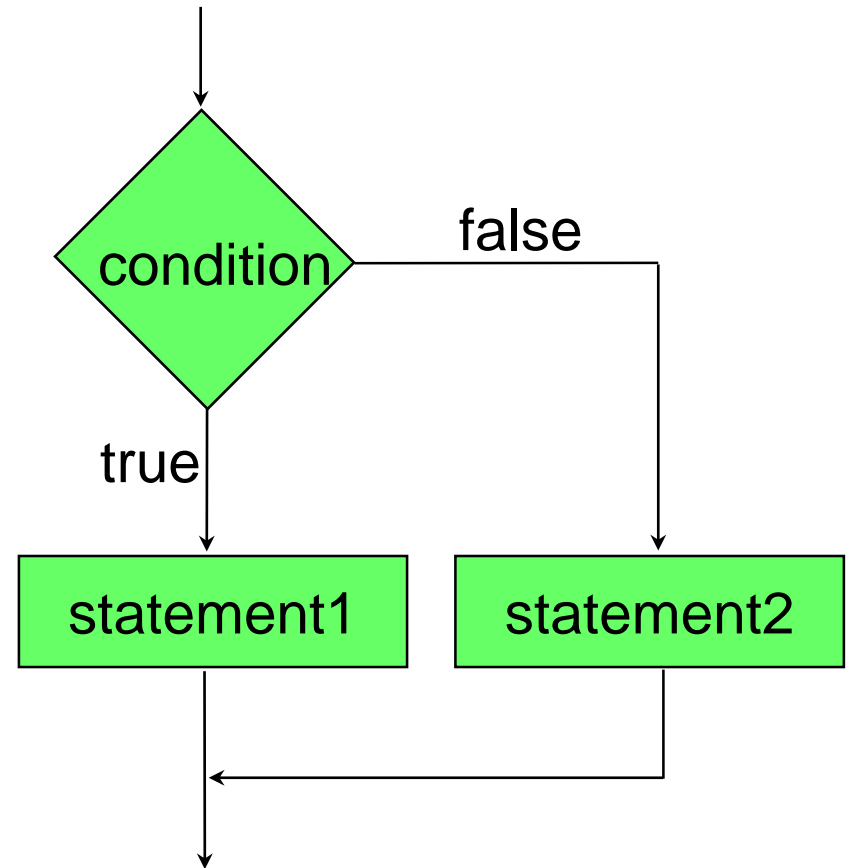
Η εμβέλεια του **y**

Μέσα στο μπλε **μπορούμε** να χρησιμοποιήσουμε την μεταβλητή **y**, αλλά **δεν** μπορούμε να **ορίσουμε** άλλη μεταβλητή με το όνομα **y**

Κάθε block έχει το δικό του χώρο μνήμης. Σε ένα χώρο μνήμης μια μεταβλητή μπορεί να οριστεί μόνο μία φορά. Ο χώρος μνήμης ενός block περιλαμβάνει και τα φωλιασμένα blocks.

To if-else statement

- Το if-else statement δουλεύει καλά όταν στο condition θέλουμε να περιγράψουμε μια επιλογή με **δύο** πιθανά ενδεχόμενα.
- Τι γίνεται αν η συνθήκη μας έχει πολλά ενδεχόμενα?



Παράδειγμα

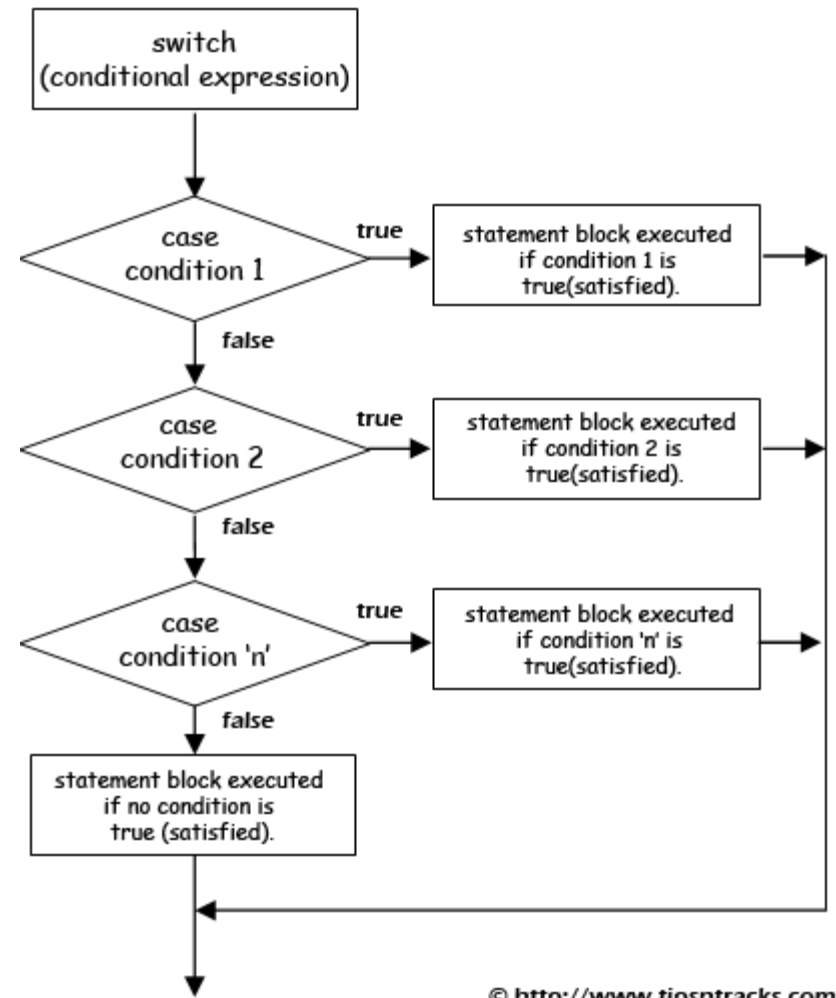
- Ένα πρόγραμμα που να εύχεται καλημέρα σε τρεις διαφορετικές γλώσσες ανάλογα με την επιλογή του χρήστη.

Παράδειγμα

- Ένα πρόγραμμα που να εύχεται καλημέρα σε τρεις διαφορετικές γλώσσες ανάλογα με την επιλογή του χρήστη.

Switch statement

```
switch (<condition expression>) {  
  case <condition 1>:  
    code statements 1  
    break;  
  case <condition 2>:  
    code statements 2  
    break;  
  case <condition 3>:  
    code statements 3  
    break;  
  default:  
    default statements  
    break;  
}
```



© <http://www.tipsntracks.com>

- **case**: οι διάφορες περιπτώσεις/τιμές που θέλουμε να ελέγξουμε
- Ο έλεγχος ροής γίνεται με τα **break**. Αν δεν υπάρχει το **break** τότε εκτελείται όλος ο κώδικας που ακολουθεί το **case**.
- **default**: Κώδικας για την περίπτωση που κανένα **case** δεν ικανοποιείται

```
import java.util.Scanner;

class SwitchTest{
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        String option = input.next();

        switch(option){
            case "GR": // if (option.equals("GR"))
                System.out.println("kalimera");
                break;
            case "EN": // if (option.equals("EN"))
                System.out.println("good morning");
                break;
            case "FR": // if (option.equals("FR"))
                System.out.println("bonjour");
                break;
            default:
                System.out.println("I do not speak this language.\n" +
                    "Greek, English, French only");
        }
    }
}
```

Αν θέλουμε να μπορούμε να απαντάμε και με μικρά?

```
import java.util.Scanner;

class SwitchTest{
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        String option = input.next();

        switch(option) {
            case "GR":
            case "gr":
                System.out.println("kalimera");
                break;
            case "EN":
            case "en":
                System.out.println("good morning");
                break;
            case "FR":
            case "fr":
                System.out.println("bonjour");
                break;
            default:
                System.out.println("I do not speak this language.\n" +
                    "Greek, English, French only");
        }
    }
}
```

Παράδειγμα

- Ένα πρόγραμμα που να διαλέγεις μια κουρτίνα και να σου δείχνει τι υπάρχει από πίσω

```
import java.util.Scanner;

class OtherSwitchTest
{
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Pick a curtain");
        int option = input.nextInt();
        switch (option)
        {
            case 1:
                System.out.println("You selected curtain 1. Zong!");
                break;
            case 2:
                System.out.println(
                    "You selected curtain 2. Congratulations!");
                break;
            case 3:
                System.out.println("You selected curtain 3. Zong!");
                break;
            default:
                System.out.println("Zong!");
        }
    }
}
```

3. JAVA ΚΛΑΣΕΙΣ

Κλάσεις και αντικείμενα στην Java

Strings, Wrapper Classes

Δομές: Πίνακες, ArrayList, HashSet, HashMap

Κλάση

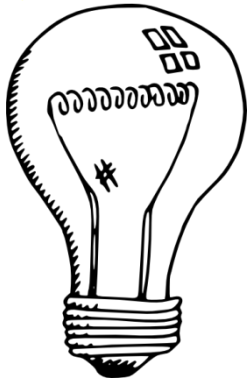
- Μια **κλάση** είναι μία αφηρημένη περιγραφή αντικειμένων με κοινά **χαρακτηριστικά** και κοινή **συμπεριφορά**.
 - Ένα **καλούπι/πρότυπο** που παράγει αντικείμενα
- Ένα **αντικείμενο** είναι ένα **στιγμιότυπο** μίας κλάσης.
- Η κλάση ορίζει τον **τύπο** του αντικειμένου.
 - Τα **χαρακτηριστικά** του αντικειμένου
 - Τις **ενέργειες** που μπορεί να επιτελέσει.

Πρακτικά στον κώδικα

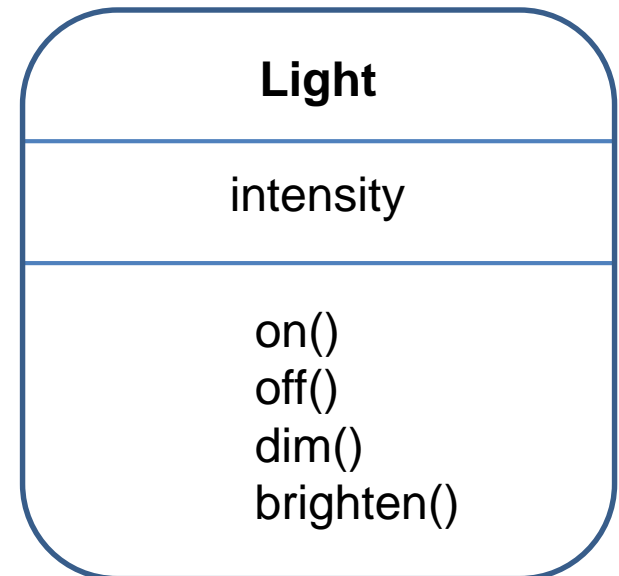
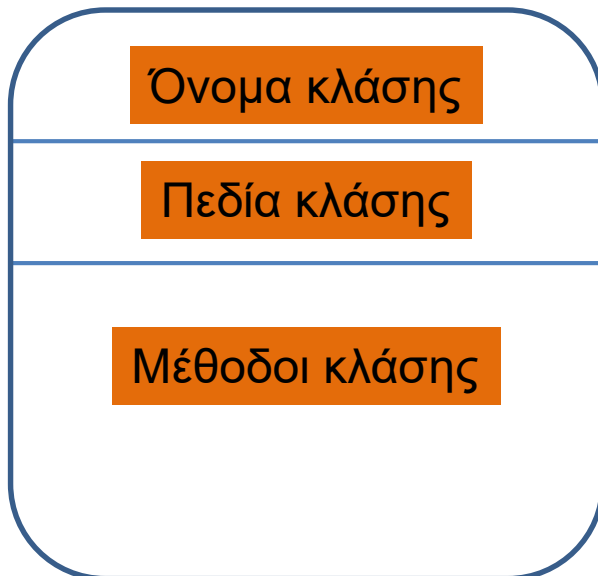
- Μία κλάση **K** ορίζεται από
 - Κάποιες **μεταβλητές** τις οποίες ονομάζουμε **πεδία**
 - Κάποιες **συναρτήσεις** που τις ονομάζουμε **μεθόδους**.
 - Οι μέθοδοι «**βλέπουν**» τα πεδία της κλάσης
- Ένα **αντικείμενο** ορίζεται ως μια **μεταβλητή τύπου K**
 - Το αντικείμενο έχει συγκεκριμένες **τιμές** στα πεδία.
 - Στο πρόγραμμα έχουμε (συνήθως) **πρόσβαση** μόνο τις **μεθόδους**.
 - Μέσω των μεθόδων έχουμε πρόσβαση στα πεδία
 - Αν υπάρχουν κάποια **πεδία** στα οποία έχουμε πρόσβαση αυτά τα λέμε **properties**.

} μέλη
της
κλάσης

Γενική μορφή της κλάσης



Έχουμε μια κλάση η οποία υλοποιεί το φως σε ένα δωμάτιο, το οποίο έχει και ένα dimmer



Δημιουργία αντικειμένων

- Δημιουργία αντικειμένου:

```
<Όνομα Κλάσης> myObject = new <Όνομα Κλάσης> ([Ορίσματα]);
```

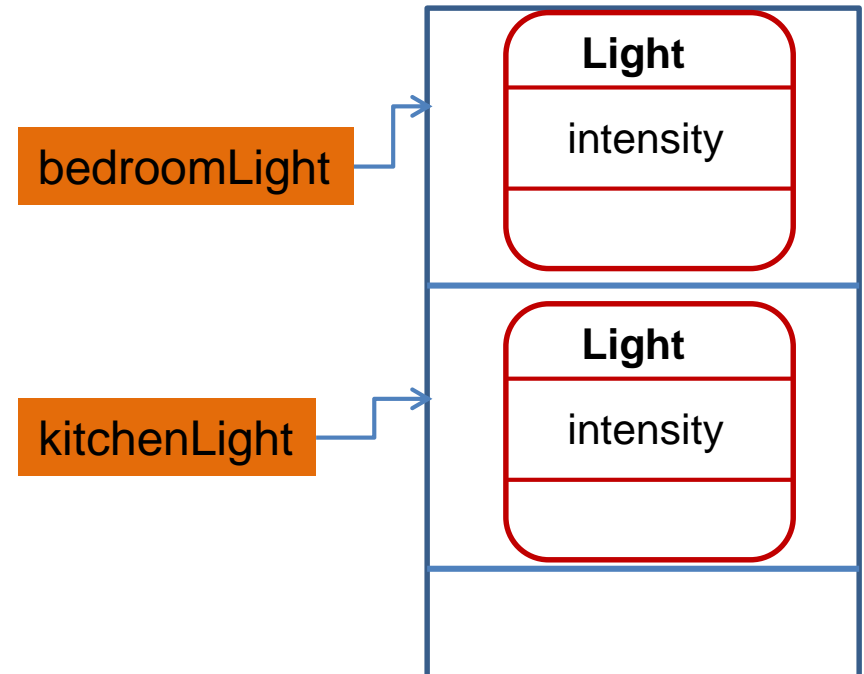
- Η λέξη κλειδί **new** δημιουργεί ένα καινούριο αντικείμενο
 - Δεσμεύει τον απαραίτητο χώρο στην μνήμη.
- Παράδειγμα που έχουμε δει ήδη:
`Scanner input = new Scanner(System.in)`
- Η λίστα των ορισμάτων μπορεί να είναι και κενή

Δημιουργία αντικειμένων

```
Light bedroomLight = new Light();
```

```
Light kitchenLight = new Light();
```

- Με την εντολή **new** δημιουργούμε ένα καινούριο αντικείμενο της κλάσης και του δίνουμε ένα **όνομα**.
- Η **new** δεσμεύει **χώρο μνήμης** για το αντικείμενο
 - Μας επιστρέφει την **διεύθυνση** του χώρου που δεσμεύτηκε.
- Η **μεταβλητή** που ορίζουμε κρατάει την διεύθυνση που “**δείχνει**” σε αυτό τον χώρο μνήμης



Κλήση μεθόδων

- Η πρόσβαση που έχουμε στα αντικείμενα είναι (κατά κύριο λόγο) μέσα από τις μεθόδους τους.
- Η κλήση μιας μεθόδου
 - `<όνομα αντικειμένου>.<όνομα μεθόδου>`
- Π.χ.

```
Light bedroomLight = new Light();  
bedroomLight.on();  
bedroomLight.brighten();  
bedroomLight.dim();  
bedroomLight.off();
```

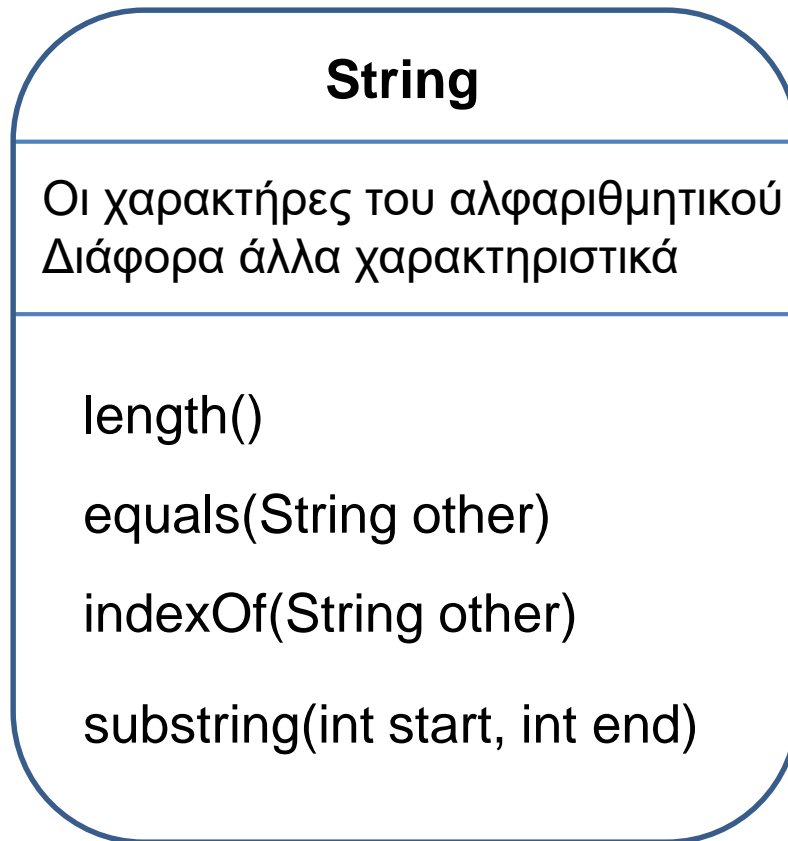
Κλήση μεθόδων

- Για να καλέσουμε μια μέθοδο μιας κλάσης θα πρέπει να δημιουργήσουμε ένα **αντικείμενο** της κλάσης
- Εξαίρεση: οι **στατικές** μέθοδοι
 - Μπορούν να κληθούν χρησιμοποιώντας το **όνομα της κλάσης**
- Παράδειγμα:
 - Η μέθοδος **main** που έχουμε δει καλείται χωρίς να έχουμε δημιουργήσει αντικείμενο
 - Γι αυτό και πρέπει να οριστεί ως **static**.
 - **System.out.println("hello") ;** καλεί την στατική μέθοδο **println** του αντικειμένου **System.out**

ΥΠΑΡΧΟΥΣΕΣ ΚΛΑΣΕΙΣ

Strings

- Έχουμε ήδη χρησιμοποιήσει κλάσεις και αντικείμενα όταν χρησιμοποιούμε Strings



Η ακριβής αναπαράσταση του αλφαριθμητικού δεν έχει και τόσο σημασία εφόσον εμείς χρησιμοποιούμε μόνο τις μεθόδους.

String αντικείμενα

- Ένα String αντικείμενο είναι μια μεταβλητή τύπου String.
 - Τρεις διαφορετικοί τρόποι να δώσουμε τιμή σε ένα String object

```
import java.util.Scanner;

class StringExample{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);

        String x = input.next();
        String z = new String("java");
        String y = "java";
    }
}
```

String μέθοδοι

- Υπάρχουν πολλές χρήσιμες μέθοδοι της κλάσης String.
 - `length()`: μήκος του String
 - `equals(String x)`: τσεκάρει για ισότητα του String που καλεί την μέθοδο με το String x
 - `trim()`: αφαιρεί κενά στην αρχή και το τέλος του string.
 - `split(char delim)`: σπάει το string σε πίνακα από strings με βάσει τον χαρακτήρα delim.
 - `indexOf(String s)`: Επιστρέφει την θέση της πρώτης εμφάνισης του s μέσα στο String που καλεί την μέθοδο
 - `substring(int start, int end)`: Επιστρέφει το υπο-string μέσα στο String που καλεί την μέθοδο μεταξύ των θέσεων start και end
 - Κλπ.

Παράδειγμα

```
class StringExample{
    public static void main(String[] args){
        String x = new String("introduction to java programming");
        String y = "java";

        int offset = x.indexOf(y);
        int end = x.length();
        x = x.substring(offset,end);
        System.out.println(x);
    }
}
```

Τα Strings είναι **αμετάβλητα** (**immutable**) αντικείμενα
Η τελευταία ανάθεση δημιουργεί ένα **καινούριο**
αντικείμενο και το αναθέτει στην μεταβλητή x

Αμετάβλητα αντικείμενα

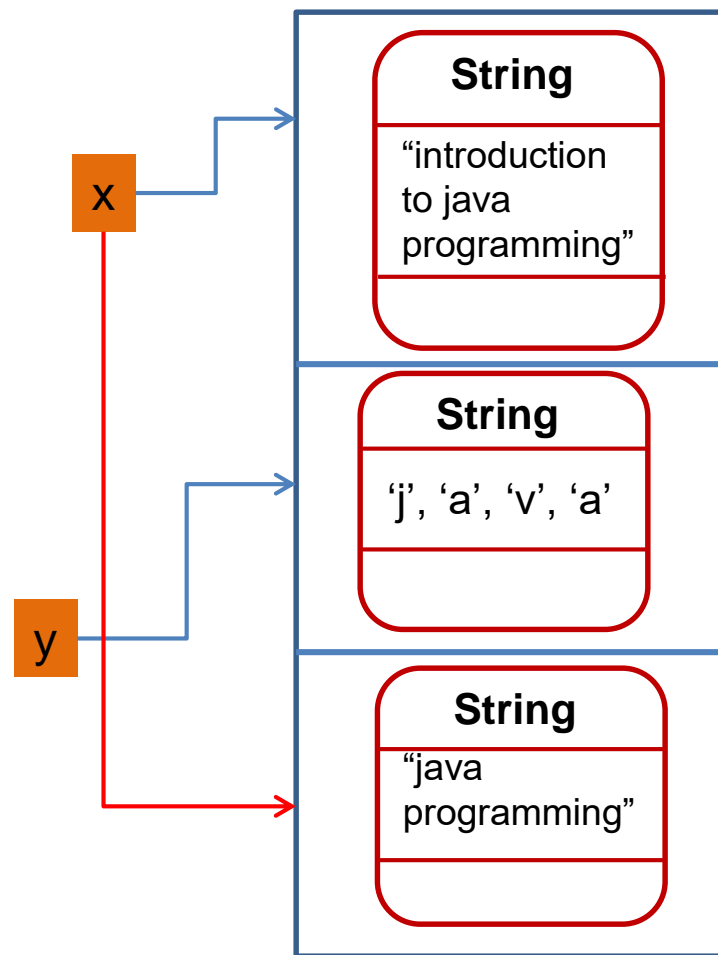
- Τα **αμετάβλητα** αντικείμενα (**immutable objects**) είναι αντικείμενα των οποίων η **εσωτερική κατάσταση** (ουσιαστικά τα πεδία τους) δεν μπορεί να μεταβληθεί.
- Τα **Strings** είναι **αμετάβλητα** αντικείμενα
 - Αυτό σημαίνει ότι δεν μπορούμε να αλλάξουμε τα περιεχόμενα ενός αντικειμένου String
 - Π.χ., δεν μπορούμε να αλλάξουμε ένα χαρακτήρα ενός String
 - Ότι αλλαγή κάνουμε έχει αποτέλεσμα να δημιουργείται ένα καινούριο String και να εκχωρείται στην μεταβλητή μας.

Αμετάβλητα αντικείμενα

```
String x = new String("introduction to java programming");  
String y = "java";  
x = x.substring(offset, end);
```

Τα Strings είναι **αμετάβλητα** (**immutable**) αντικείμενα

Η τελευταία ανάθεση δημιουργεί ένα **καινούριο** αντικείμενο και το αναθέτει στην μεταβλητή x



String σταθερές

- Οι **String τιμές** είναι κι αυτές αντικείμενα και μπορούμε να καλέσουμε τις μεθόδους τους

```
import java.util.Scanner;

class StringConstants{
    public static void main(String[] args){

        int offset = "java programming".indexOf("pro");
        int end = "java programming".length();
        String z = "java programming".substring(offset,end);
        System.out.println(z);

    }
}
```

Scanner

- Δημιουργία αντικειμένου Scanner
 - `Scanner input = new Scanner(System.in) ;`
- Μέθοδοι της Scanner:
 - `next ()` : επιστρέφει το επόμενο String από την είσοδο (όλοι οι χαρακτήρες από το σημείο που σταμάτησε την προηγούμενη φορά μέχρι να βρει white space: κενό, tab, αλλαγή γραμμής)
 - `nextInt ()` : διαβάζει το επόμενο String και το μετατρέπει σε int και επιστρέφει ένα int αριθμό.
 - `nextDouble ()` : διαβάζει το επόμενο String και το μετατρέπει σε double και επιστρέφει τον double αριθμό.
 - `nextLine ()` : Διαβάζει ότι υπάρχει μέχρι να βρει newline και το επιστρέφει ως String.

Wrapper classes

- Για κάθε βασικό τύπο η Java έχει και μία **wrapper class**:
 - **Integer** class
 - **Double** class
 - **Boolean** class
- Οι κλάσεις αυτές έχουν κάποιες μεθόδους και πεδία που μπορεί να μας είναι χρήσιμα
 - Κατά κύριο λόγο **μετατροπή** από και προς **string**
 - Τη **μέγιστη** και την **ελάχιστη** τιμή κάθε τύπου
- Κάποιες από τις μεθόδους είναι **στατικές**
 - Μπορούμε να τις καλέσουμε **χωρίς να έχουμε αντικείμενο**.
 - Έχουμε επίσης πρόσβαση σε κάποια **στατικά πεδία**.

Παράδειγμα

```
class WrapperTest{
    public static void main(String args[])
    {
        int i = Integer.valueOf("2");
        double d = Double.parseDouble("2.5");
        System.out.println(i*d);
        Integer x = 5;
        Double y = 2.5;
        String s = x.toString() + y.toString();
        System.out.println(s);
        System.out.println(Integer.MAX_VALUE);
    }
}
```

ΔΟΜΕΣ

Πίνακες

- Πολλές φορές έχουμε πολλές μεταβλητές του ίδιου τύπου που συσχετίζονται και θέλουμε να τις βάλουμε μαζί.
 - Τα ονόματα των φοιτητών σε μία τάξη
 - Οι βαθμοί ενός φοιτητή για όλα τα εργαστήρια.
- Για το σκοπό αυτό χρησιμοποιούμε τους **πίνακες**.
- ΣΥΝΤΑΚΤΙΚΟ:
 - **Τύπος [] myArray;**
 - Ο **τύπος** είναι οποιοσδήποτε μια οποιαδήποτε κλάση. Είναι ο τύπος των δεδομένων που αποθηκεύει ο πίνακας μας.
- Παραδείγματα:
 - **int [] integerArray;** // πίνακας από ακεραίους
 - **String [] stringArray;** // πίνακας από String

Πίνακες

- Ορισμός πίνακα:

```
int[] myArray1 = {10,20};  
int[] myArray2 = new int[2];  
int[] myArray3;
```

Ορίζει ένα πίνακα ακεραίων δύο θέσεων με αρχικές τιμές 10,20

Ορίζει ένα πίνακα ακεραίων δύο θέσεων χωρίς αρχικές τιμές (αρχικοποιούνται αυτόματα στο μηδέν)

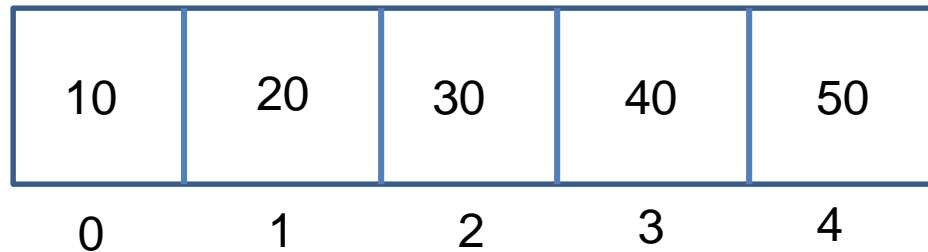
Ορίζει μια μεταβλητή που είναι πίνακας από ακεραίους αλλά δεν δεσμεύει χώρο για τον πίνακα

- Οι πίνακες ορίζονται με ένα μέγεθος (**length**) και αυτό **δεν αλλάζει** εκτός αν ορίσουμε ξανά τον πίνακα.
- Στη Java ένας πίνακας είναι ένα **αντικείμενο** και έχει properties
 - `System.out.println(myArray2.length);`
 - Τυπώνει το **μέγεθος** του πίνακα.

Πρόσβαση των στοιχείων του πίνακα

- **Προσοχή!** Τα στοιχεία του πίνακα αριθμούνται από το `0...length-1` (**ΟΧΙ** `1...length`)

- `int myArray[] = {10,20,30,40,50};`



- Για να προσπελάσουμε το **δεύτερο** στοιχείο του πίνακα
 - `myArray[1] += 5;`
 - `System.out.println(myArray[1]);`

Διατρέχοντας ένα πίνακα

- Στην Java έχουμε δύο τρόπους να διατρέχουμε ένα πίνακα

Διατρέχουμε τα στοιχεία

```
for (<array type> element: array)
{
    ... do something with element...
}
```

```
int array[] = {1,3,5,7};
for (int element: array)
{
    System.out.println(element)
}
```

Διατρέχουμε τις θέσεις του πίνακα

```
for (int i = 0; i < array.length; i ++ )
{
    ... do something with array[i]...
}
```

```
int array[] = {1,3,5,7};
for (int i = 0; i < array.length; i ++ )
{
    System.out.println(array[i])
}
```

```
class TestArrays1 {
    public static void main(String [] args)
        int[] arr0; // int arr0[];

        int[] arr1 = {1, 2, 3, 4};
        for (int i = 0; i < arr1.length; i ++){
            System.out.println(arr1[i]);
        }

        int arr2[] = new int [10];
        arr0 = arr2;
        for (int i = 0; i < arr2.length; i ++){
            arr2[i] = i+1;
        }
        for (int x: arr0){
            System.out.println(x);
        }
    }
}
```

Εναλλακτικό συντακτικό

Ορισμός πίνακα χωρίς χώρο

Ο πίνακας arr0 δείχνει στον ίδιο χώρο μνήμης όπως και ο arr2. Όποια αλλαγή γίνει στον ένα θα εμφανιστεί και στον άλλο

Παράδειγμα

- Τυπώστε όλα τα **στοιχεία** του πίνακα και όλα τα ζεύγη από **στοιχεία** στον πίνακα

```
class ScanArray
{
    public static void main(String [] args)
    {
        double [] array = {5.3, 3.4, 2.3, 1.2, 0.1};

        // Print all elements
        for (double element: array){
            System.out.println(element);
        }

        // Print all pairs of elements
        for (int i = 0; i < array.length; i ++){
            for (int j = i+1; j < array.length; j ++){
                System.out.println(array[i] + " " + array[j]);
            }
        }
    }
}
```

Σε αυτή την περίπτωση δεν μπορούμε να διατρέξουμε τα στοιχεία, πρέπει να διατρέξουμε τις θέσεις

Πολυδιάστατοι πίνακες

- Μπορούμε να ορίσουμε και **πολυδιάστατους** πίνακες

- `int[][] myArray1 = {{10,20,30},{3,4,5}};`

10	20	30
3	4	5

- `int[][] myArray2 = new int[2][3];`

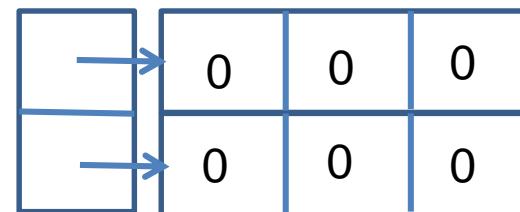
10	20	30
3	4	5

Πολυδιάστατοι πίνακες

- Ένας δισδιάστατος πίνακας είναι ένας **πίνακας από αντικείμενα-πίνακες**.

- `int[][] myArray3;`

Ο τύπος του πίνακα είναι `int[]`, δηλαδή πίνακας από `int`



- `int[][] myArray3 = new int[2][];`
- Η κάθε γραμμή είναι ένας πίνακας και πρέπει να αρχικοποιηθεί
 - `myArray3[0] = new int[3];`
 - `myArray3[1] = new int[3];`

Πολυδιάστατοι πίνακες

- Ένας δισδιάστατος πίνακας είναι ένας **πίνακας από αντικείμενα-πίνακες**.

- `int[][] myArray3;`

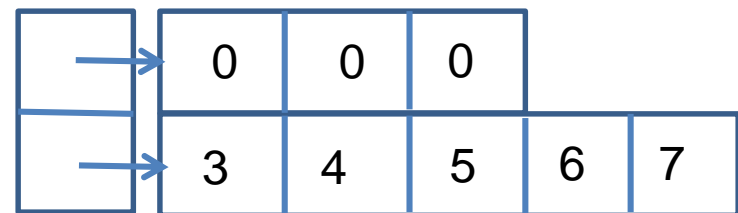
- `int[][] myArray3 = new int[2][];`

- `myArray3[0] = new int[3];`

- `myArray3[1] = new int[3];`

- Ο πίνακας μπορεί να είναι ασύμμετρος

- `myArray3[1] = {3,4,5,6,7}`



- Τι παίρνω για τα παρακάτω?

- `System.out.println(myArray3.length);`

- `System.out.println(myArray3[1].length);`

```

public class TestArrays2 {
    public static void main(String [] args) {
        int[][] arr3 = {{1, 2, 3}, {3, 4, 5}};

        int[][] arr4 = new int [10][20];

        arr4 = arr3;

        System.out.println(arr4.length + " "
            + arr4[0].length);

        int arr5[][] = new int[2][];
        arr5[0] = new int[3];
        arr5[1] = new int[5];
        System.out.println(arr5.length + " "
            + arr5[0].length + " "
            + arr5[1].length);
    }
}

```

Πίνακας με αρχικές τιμές

Πίνακας 10x20

Ο πίνακας arr4 γίνεται ίδιος με τον arr3. Δηλαδή δείχνει στον ίδιο χώρο μνήμης. Ο προηγούμενος χώρος χάνεται

Τυπώνει "2 3"

Ασύμμετρος πίνακας

Τυπώνει "2 3 5"

Αρχικοποίηση πινάκων

- Δημιουργήστε τους παρακάτω πίνακες:
 - Ένα μονοδιάστατο πίνακα με n θέσεις με τις τιμές $0..n-1$
 - Ένα δισδιάστατο πίνακα με $n \times n$ θέσεις με τις τιμές $0 \dots n^2-1$
 - Ένα κάτω διαγώνιο πίνακα $n \times n$ (π.χ. παρακάτω 3×3)
- Το μέγεθος του πίνακα θα το δίνει ο χρήστης

0		
1	2	
3	4	5

```
import java.util.Scanner;

class ArrayInitialization
{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int n = input.nextInt();

        int[] array1d = new int[n];
        for (int i = 0; i < n; i ++){
            array1d[i] = i;
        }
        for (int i = 0; i < n; i ++){
            System.out.print(array1d[i] + " ");
        }
        System.out.println();
    }
}
```



```
import java.util.Scanner;

class ArrayInitialization
{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int n = input.nextInt();

        int[][] array2d = new int[n][n];
        for (int i = 0; i < n; i ++){
            for (int j = 0; j < n; j ++){
                array2d[i][j] = i*n+j;
            }
        }
        for (int i = 0; i < n; i ++){
            for (int j = 0; j < n; j ++){
                System.out.print(array2d[i][j] + " " );
            }
            System.out.println();
        }
    }
}
```

```
import java.util.Scanner;

class ArrayInitialization
{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int n = input.nextInt();

        int[][] lowerDiagonal = new int[n][];
        for (int i = 0; i < n; i ++){
            lowerDiagonal[i] = new int[i+1];
            for (int j = 0; j < i+1; j ++){
                lowerDiagonal[i][j] = i*(i+1)/2 + j;
            }
        }
        for (int i = 0; i < n; i ++){
            for (int j = 0; j < i+1; j ++){
                System.out.print(lowerDiagonal[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Πίνακες από Strings

- Φτιάξετε ένα πρόγραμμα που να διαβάζει από την είσοδο το όνομα και το επώνυμο η ατόμων και τα αποθηκεύει. Το η δίνεται στην είσοδο

```
import java.util.Scanner;

class Names
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Give number of persons:");
        int n = input.nextInt();
        String[][] nameArray = new String[n][2];
        for (int i = 0; i < n; i++){
            nameArray[i][0] = input.next();
            nameArray[i][1] = input.next();
        }

        for (int i = 0; i < n; i++){
            System.out.println(nameArray[i][0]
                + " " + nameArray[i][1]);
        }
    }
}
```

Παράδειγμα με strings και πίνακες

- Φτιάξτε ένα πρόγραμμα που να διαβάζει μία γραμμή από κείμενο και να ψάχνει μία λέξη που δίνουμε σαν όρισμα μέσα σε αυτή τη γραμμή.

➤ `java LookFor hello`

- Περιμένει να διαβάσει μια γραμμή από κείμενο και ψάχνει τη λέξη `hello` μέσα στο κείμενο.

```
import java.util.Scanner;

class LookFor
{
    public static void main(String args[])
    {
        String name = "default";
        if (args.length == 1)
        {
            name = args[0];
        }
        Scanner input = new Scanner(System.in);
        String line = input.nextLine();
        String [] words = line.split(" ");
        for (int i =0; i < words.length; i ++)
        {
            if (name.equals(words[i])) {
                System.out.println(name + " found it at " + i);
            }
        }
    }
}
```

Τα command-line ορίσματα του προγράμματος αποθηκεύονται στον **πίνακα** από Strings που είναι όρισμα στην main()

Η μέθοδος split της κλάσης String με όρισμα ένα delimiter string σπάει το String με βάση το delimiter και επιστρέφει ένα πίνακα από Strings

Στην περίπτωση αυτή σπάμε το line με βάση το κενό και παίρνουμε τις λέξεις.

Η κλάση ArrayList

- Η κλάση `ArrayList` ορίζει ένας **δυναμικό πίνακα** με **μεταβλητό μέγεθος** ανάλογα με τον αριθμό των στοιχείων που τοποθετούμε
 - Παρόμοιο με το `list` στην Python
 - Το `ArrayList` κρατάει **αντικείμενα** συγκεκριμένου τύπου.
- ΣΥΝΤΑΚΤΙΚΟ:
 - `import java.util.ArrayList;`
 - `ArrayList<Τύπος> myList;`
- Ο **ΤΥΠΟΣ** είναι οποιοσδήποτε μια οποιαδήποτε κλάση.
 - Αυτός είναι ο τύπος των δεδομένων που αποθηκεύει ο πίνακας μας.
 - Για να αποθηκεύσουμε **βασικούς τύπους** χρειαζόμαστε την **wrapper class**.
- Παραδείγματα:
 - `ArrayList<Integer> myList; // λίστα από ακεραίους`
 - `ArrayList<String> myList; // λίστα από String`

ArrayList

- Δημιουργία αντικειμένου
 - `ArrayList<T> myList = new ArrayList<T>();`
- Μέθοδοι
 - `size()`: ο αριθμός των στοιχείων του πίνακα.
 - `add(T x)`: προσθέτει το στοιχείο `x` στο τέλος του πίνακα.
 - `add(int i, T x)`: προσθέτει το στοιχείο `x` στη θέση `i` και μετατοπίζει τα υπόλοιπα στοιχεία κατά μια θέση.
 - `set(int i, T x)`: θέτει στην θέση `i` την τιμή `x` αλλάζοντας την προηγούμενη
 - `get(int i)`: επιστρέφει το αντικείμενο τύπου `T` στη θέση `i`.
 - `remove(int i)`: αφαιρεί το στοιχείο στη θέση `i`
 - `remove(T x)`: αφαιρεί την πρώτη εμφάνιση του `x`
 - `contains(T x)`: επιστρέφει `true/false` αν το `x` περιέρχεται στην λίστα
- Διατρέχοντας τον πίνακα:
 - `ArrayList<T> myList = new ArrayList<T>();`
 - `for(T x: myList){...}`

Παράδειγμα

- Ζητάμε από την είσοδο θετικούς ακεραίους αριθμούς μέχρι ο χρήστης να δώσει το -1. Αποθηκεύστε τους αριθμούς σε ένα πίνακα και τυπώστε τους
- Δεν ξέρουμε εκ των προτέρων πόσους αριθμούς θα πρέπει να αποθηκεύσουμε.
 - Θα χρησιμοποιήσουμε ArrayList αντί για πίνακα.

```
import java.util.ArrayList;
import java.util.Scanner;

class ArrayListTest
{
    public static void main(String[] args){
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        Scanner input = new Scanner(System.in);
        int x = input.nextInt();
        while (x != -1){
            numbers.add(x);
            x = input.nextInt();
        }

        for (Integer y: numbers){
            System.out.print(y + " ");
        }
        System.out.println();
    }
}
```

Πίνακες από Strings II

- Φτιάξετε ένα πρόγραμμα που να διαβάζει από την είσοδο όνομα και επώνυμο μέχρι να τερματίσουμε την είσοδο.

```
import java.util.Scanner;
import java.util.ArrayList;

class Names2
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        ArrayList<String[]> nameList =
            new ArrayList<String[]>();
        while (input.hasNext()) {
            String[] nameArray = new String[2];
            nameArray[0] = input.next();
            nameArray[1] = input.next();
            nameList.add(nameArray);
        }

        for (String[] nameArray:nameList) {
            System.out.println(nameArray[0]
                + " " + nameArray[1]);
        }
    }
}
```

Ορισμός λίστας
από πίνακες
String

Σύνολα

- Οι πίνακες και η κλάση `ArrayList` μας επιτρέπουν να κρατάμε μια **διατεταγμένη** σειρά από αντικείμενα
- Αν δεν μας ενδιαφέρει η διάταξη, αλλά μόνο η συλλογή των αντικειμένων μπορούμε να τα αποθηκεύσουμε σε ένα **σύνολο**.
- Στην Java μια κλάση για να ορίσουμε ένα σύνολο από αντικείμενα είναι η κλάση **`HashSet`**
- Τα σύνολα μας επιτρέπουν **γρήγορη αναζήτηση** μιας τιμής μέσα στην συλλογή.
- ΣΥΝΤΑΚΤΙΚΟ:
 - `import java.util.HashSet;`
 - `HashSet<Τύπος> mySet;`

HashSet

- Δημιουργία αντικειμένου
 - `HashSet<T> mySet = new HashSet<T>();`
- Μέθοδοι
 - `size()` : ο αριθμός των στοιχείων στο σύνολο.
 - `add(T x)` : προσθέτει το στοιχείο `x` αν δεν υπάρχει ήδη στο σύνολο.
 - `remove(T x)` : αφαιρεί το στοιχείο `x`.
 - `contains(T x)` : επιστρέφει `true/false` αν το σύνολο περιέχει το στοιχείο `x`.
 - `isEmpty()` : boolean αν έχει στοιχεία το σύνολο ή όχι.
- Διατρέχοντας τα στοιχεία του συνόλου:
 - `HashSet<T> mySet = new HashSet<T>();`
 - `for(T x: mySet) {...}`

Παράδειγμα I

- Διαβάζουμε ένα String που περιέχει ένα κείμενο και θέλουμε να βρούμε όλες τις μοναδικές λέξεις στο κείμενο
 - Π.χ. να φτιάξουμε το λεξικό ενός βιβλίου

```
import java.util.HashSet;
import java.util.Scanner;

public class HashSetExample
{
    public static void main(String[] args) {
        HashSet<String> mySet = new HashSet<String>();
        Scanner input = new Scanner(System.in);

        String line = input.nextLine();
        String words[] = line.split(" ");
        for (String word: words) {
            if (!mySet.contains(word)) {
                mySet.add(word);
            }
        }

        for (String word: mySet) {
            System.out.println(word);
        }
    }
}
```

Δήλωση μιας μεταβλητής
HashSet από Strings.

Τοποθετούμε στο HashSet
μόνο τα Strings τα οποία δεν
έχουμε ήδη δει (δεν είναι ήδη
στο σύνολο)

Διατρέχουμε και να τυπώνουμε τα
στοιχεία του HashSet


```
import java.util.HashSet;
import java.util.Scanner;

public class HashSetExample
{
    public static void main(String[] args) {
        HashSet<String> mySet = new HashSet<String>();
        Scanner input = new Scanner(System.in);

        String text = input.nextLine();
        String words[] = text.split(" ");
        for (String word: words) {
            mySet.add(word);
        }

        for (String word: mySet) {
            System.out.println(word);
        }
    }
}
```

Επειδή το HashSet κρατάει **μοναδικά** αντικείμενα, δεν χρειάζεται να κάνουμε τον έλεγχο. Αν υπάρχει ήδη το String δεν θα το ξαναπροθέσει.

Λεξικά

- Οι πίνακες και η κλάση `ArrayList` μας επιτρέπουν να κρατάμε μια διατεταγμένη σειρά από αντικείμενα και τα **δεικτοδοτούν** με τη **θέση** τους.
- Αν θέλουμε να δεικτοδοτήσουμε με κάποιο **κλειδί** χρειαζόμαστε **λεξικό**.
- Τα λεξικά κρατάνε ζευγάρια της μορφής **(κλειδί,τιμή)** και επιτρέπουν **γρήγορη αναζήτηση** με το κλειδί
- ΣΥΝΤΑΚΤΙΚΟ
 - `import java.util.HashMap;`
 - `HashMap<Τύπος Κλειδιου, Τύπος Τιμής> myMap;`

HashMap

- Δημιουργία αντικειμένου
 - `HashMap<K, V> myMap = new HashMap<K, V> ();`
- Μέθοδοι
 - `size ()` : ο αριθμός των στοιχείων (κλειδιών) στο map.
 - `put (K key, V value)` : προσθέτει το ζευγάρι `(key, value)` (δημιουργεί μία συσχέτιση)
 - `V get (K key)` : επιστρέφει την τιμή για το κλειδί `key`.
 - `remove (K key)` : αφαιρεί το ζευγάρι με κλειδί `key`.
 - `containsKey (K key)` : boolean αν το σύνολο περιέχει το κλειδί `key` ή όχι.
 - `containsValue (V value)` : boolean αν το σύνολο περιέχει την τιμή `value` ή όχι. (αργό)
 - `isEmpty ()` : boolean αν έχει στοιχεία το map ή όχι.
 - `Set<K> keySet ()` : επιστρέφει ένα `Set` με τα κλειδιά.

Παράδειγμα II

- Διαβάζουμε ένα String που αναπαριστά ένα κείμενο και θέλουμε να βρούμε όλες τις μοναδικές λέξεις μέσα στο κείμενο και τον αριθμό των εμφανίσεων τους.
- Πώς θα το υλοποιήσουμε αυτό?

```
import java.util.HashMap;
import java.util.Scanner;
```

```
class HashMapExample
{
```

```
    public static void main(String[] args) {
```

```
        HashMap<String, Integer> myMap =
            new HashMap<String, Integer>();
```

```
        Scanner input = new Scanner(System.in);
```

```
        String text = input.nextLine();
```

```
        String words[] = text.split(" ");
```

```
        for (String word: words) {
```

```
            if (!myMap.containsKey(word)) {
                myMap.put(word, 1);
```

```
            }else{
```

```
                myMap.put(word, myMap.get(word) + 1);
```

```
            }
```

```
        for (String word: myMap.keySet()) {
```

```
            System.out.println(word + " : " + myMap.get(word));
```

```
        }
```

```
    }
```

```
}
```

Δήλωση μιας μεταβλητής **HashMap** που συσχετίζει **Strings** (κλειδιά) και **Integers** (τιμές). Για κάθε λέξη (String) τον αριθμό εμφανίσεων (Integer)

Αν η λέξη δεν είναι ήδη στο HashMap τότε πρόσθεσε την με τιμή 1.

Αλλιώς πάρε την τιμή της λέξης, αύξησε την κατά 1, και βάλε τη πίσω σαν νέα τιμή

Διέτρεξε το σύνολο με τα κλειδιά (ονόματα) στο HashMap
Για κάθε κλειδί πάρε και τύπωσε την τιμή του.

4. ΔΗΜΙΟΥΡΓΙΑ ΚΛΑΣΕΩΝ ΚΑΙ ΑΝΤΙΚΕΙΜΕΝΩΝ

Ορισμός κλάσεων και αντικειμένων

Πεδία, Μέθοδοι, παράμετροι,
επιστρεφόμενες τιμές, τοπικές μεταβλητές

Constructor, Accessor, Mutator μεθοδοι

Κλάση

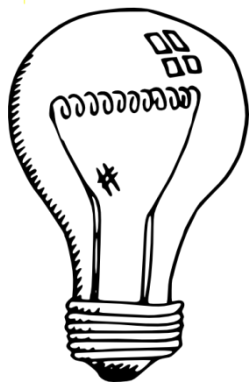
- Μια **κλάση** είναι μία αφηρημένη περιγραφή αντικειμένων με κοινά **χαρακτηριστικά** και κοινή **συμπεριφορά**.
 - Ένα **καλούπι/πρότυπο** που παράγει αντικείμενα
- Ένα **αντικείμενο** είναι ένα **στιγμιότυπο** μίας κλάσης.
- Η κλάση ορίζει τον **τύπο** του αντικειμένου.
 - Τα **χαρακτηριστικά** του αντικειμένου
 - Τις **ενέργειες** που μπορεί να επιτελέσει.

Πρακτικά στον κώδικα

- Μία κλάση **K** ορίζεται από
 - Κάποιες **μεταβλητές** τις οποίες ονομάζουμε **πεδία**
 - Κάποιες **συναρτήσεις** που τις ονομάζουμε **μεθόδους**.
 - Οι μέθοδοι «**βλέπουν**» τα πεδία της κλάσης
- Ένα **αντικείμενο** ορίζεται ως μια **μεταβλητή τύπου K**
 - Το αντικείμενο έχει συγκεκριμένες **τιμές** στα πεδία.
 - Στο πρόγραμμα έχουμε (συνήθως) **πρόσβαση** μόνο τις **μεθόδους**.
 - Μέσω των μεθόδων έχουμε πρόσβαση στα πεδία
 - Αν υπάρχουν κάποια **πεδία** στα οποία έχουμε πρόσβαση αυτά τα λέμε **properties**.

} μέλη
της
κλάσης

Δημιουργώντας φως



Θα φτιάξουμε μια κλάση που θα χειρίζεται ένα διακόπτη φωτός. Το φως είναι είτε ανοιχτό είτε κλειστό και μπορούμε να ανοιγοκλείνουμε το φως

Όνομα κλάσης

Πεδία κλάσης

Μέθοδοι κλάσης

Light

boolean lightIsOn

flipSwitch()

Κλάσεις και αντικείμενα

- Ορισμός κλάσης:

```
class <Όνομα Κλάσης>
{
    <Ορισμός πεδίων κλάσης>

    <Ορισμός μεθόδων κλάσης>
}
```

- Ορισμός αντικειμένου:

```
<Όνομα Κλάσης> myObject = new <Όνομα Κλάσης>();
```

- Ο ορισμός του αντικειμένου γίνεται συνήθως μέσα στη **main** ή μέσα στη μέθοδο μίας **άλλης κλάσης** που χρησιμοποιεί το αντικείμενο

Ορισμός μελών κλάσης

- Ορισμός πεδίων

- Τα πεδία είναι μεταβλητές και ορίζονται όπως οι υπόλοιπες μεταβλητές
 - Η μόνη διαφορά είναι ότι τα πεδία πρέπει να τα προσδιορίσουμε ως `private` ή `public`. Τα πεδία τα ορίζουμε πάντα `private`.

```
[private/public] <τύπος> <όνομα μεταβλητής> [ = τιμή ] ;
```

- Ορισμός μεθόδων

- Οι μέθοδοι είναι συναρτήσεις και έχουν το εξής ορισμό

```
[public/private] <τύπος> <όνομα συνάρτησης> ( [παράμετροι] )  
{  
    <κώδικας συνάρτησης>  
}
```

Light

```
class Light
```

```
{
```

```
    private boolean lightIsOn = false;
```

```
    public void flipSwitch()
```

```
    {
```

```
        lightIsOn = !lightIsOn;
```

```
    }
```

```
}
```

```
class HouseWithLights
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Light bedroomLight = new Light();
```

```
        bedroomLight.flipSwitch();
```

```
    }
```

```
}
```

Ορισμός κλάσης

Ορισμός
(και αρχικοποίηση) πεδίου

Ορισμός μεθόδου τύπου
void χωρίς παραμέτρους

Χρήση πεδίου

Ορισμός αντικειμένου

Κλήση μεθόδου

Τα keywords Public/Private

- Ότι είναι ορισμένο ως **public** σε μία κλάση **είναι προσβάσιμο** από μία άλλη κλάση που ορίζει ένα αντικείμενο τύπου Light
 - Π.χ., η μέθοδος `flipSwitch()` **είναι προσβάσιμη** από την κλάση `HouseWithLights` μέσω του αντικειμένου `bedroomLight`.
- Ότι είναι ορισμένο ως **private** σε μία κλάση **δεν είναι προσβάσιμο** από μία άλλη κλάση
 - Π.χ., το πεδίο `lightsOn` **δεν είναι προσβάσιμο** από την κλάση `HouseWithLights` μέσω του αντικειμένου `bedroomLight`.
- Μπορούμε να έχουμε **public** και **private** πεδία και μεθόδους.
 - Κανόνας: Τα **πεδία** τα ορίζουμε (σχεδόν) **ΠΑΝΤΑ private**.
 - Οι μέθοδοι που χρειάζονται να καλούνται από **αντικείμενα** είναι **public** αυτές που είναι **βοηθητικές** είναι **private**.
- Τα πεδία και οι μέθοδοι μίας κλάσης, ανεξάρτητα αν είναι **public** ή **private**, είναι **προσβάσιμα** από όλες τις μεθόδους και τα αντικείμενα **της ίδιας κλάσης**
 - Π.χ., το πεδίο `lightsOn` είναι προσβάσιμο παντού μέσα στην κλάση `Light`, και σε οποιοδήποτε άλλο αντικείμενο τύπου `Light`
 - Κάποιοι για να ξεχωρίζουν τα πεδία από άλλες μεταβλητές βάζουν `'_'` στην αρχή του ονόματος των πεδίων. Π.χ., `_lightsOn`

```
class Light
{
    private boolean lightIsOn = false;

    public void flipSwitch() {
        lightIsOn = !lightIsOn;
    }

    public void printState() {
        if (lightIsOn) {
            System.out.println("The light is on");
        } else {
            System.out.println("The light is off");
        }
    }
}
```

Η κατάσταση ενός αντικειμένου προσδιορίζεται από τις τιμές που έχουν τα πεδία της κλάσης

```
class HouseWithLights
{
    public static void main(String[] args) {
        Light bedroomLight = new Light();
        bedroomLight.flipSwitch();
        bedroomLight.printState();
        Light kitchenLight = new Light();
        kitchenLight.flipSwitch();
        kitchenLight.printState();
    }
}
```

Η μόνη πρόσβαση που έχουμε στην κατάσταση του αντικειμένου είναι μέσω των μεθόδων της κλάσης.

Dimmer

- Θέλουμε ο διακόπτης μας να μας δίνει την δυνατότητα να αυξομειώνουμε την ένταση.
 - Τι επιπλέον πεδία πρέπει να προσθέσουμε?
 - Τι επιπλέον μεθόδους χρειαζόμαστε?

```
class DimmerLight
```

```
{
```

```
    private boolean lightIsOn = false;
```

```
    private int intensity = 100;
```

```
    public void flipSwitch(){  
        lightIsOn = !lightIsOn;  
    }
```

```
    public void dim(){  
        if (intensity > 0){  
            intensity --;  
        }  
    }
```

```
    public void brighten(){  
        if (intensity < 100){  
            intensity ++;  
        }  
    }
```

```
    public void printState(){  
        if (lightIsOn){  
            System.out.println("The light is ON with intensity " + intensity);  
        }else{  
            System.out.println("The light is OFF");  
        }  
    }
```

```
}
```

```
class HouseWithDimmerLights
```

```
{
```

```
    public static void main(String[] args){
```

```
        DimmerLight bedroomLight =  
            new DimmerLight();
```

```
        bedroomLight.flipSwitch();
```

```
        bedroomLight.dim();
```

```
        bedroomLight.printState();
```

```
    }
```

```
}
```


Παράδειγμα

- Δημιουργήστε μια κλάση η οποία μοντελοποιεί ένα χρονόμετρο που κρατάει λεπτά και δευτερόλεπτα. Το χρονόμετρο δεν μπορεί να μετρήσει πάνω από μία ώρα, μετά μηδενίζει. Υπάρχει μια μέθοδος `tick` η οποία προχωράει το χρονόμετρο κατά ένα δευτερόλεπτο. Θέλουμε επίσης και μια μέθοδο `reset` και μία που μας τυπώνει την κατάσταση του ρολογιού.

```
class Chronometer
{
    private int seconds = 0;
    private int minutes = 0;

    public void tick()
    {
        if (seconds < 59){
            seconds ++;
        }else if (minutes < 59){
            seconds = 0;
            minutes ++;
        }else{
            seconds = 0;
            minutes = 0;
        }
    }

    public void reset()
    {
        seconds = 0;
        minutes = 0;
    }

    public void printTimePassed(){
        System.out.println(minutes + " minutes and " + seconds + " seconds");
    }
}

class ChronoTest
{
    public static void main(String[] args)
    {
        Chronometer timer = new Chronometer();
        for (int i = 0; i < 100; i ++){
            timer.tick();
        }
        timer.printTimePassed();
    }
}
```

Dimmer

- Κάθε φορά που αυξάνουμε ή μειώνουμε την ένταση θέλουμε να μας λέει και την κατανάλωση
 - (Κατανάλωση = ένταση * 0.1 λεπτά/ώρα)

```
class DimmerLight2
```

```
{  
    private boolean lightIsOn = false;  
    private int intensity = 100;  
  
    public void flipSwitch(){  
        lightIsOn = !lightIsOn;  
    }  
  
    public void dim(){  
        if (intensity > 0){  
            intensity --;  
            double consumption = intensity *0.1;  
            System.out.print("Consumption = "+consumption);  
        }  
  
    public void brighten(){  
        if (intensity < 100){  
            intensity ++;  
            double consumption = intensity *0.1;  
            System.out.print("Consumption = "+consumption);  
        }  
  
    public void printState(){  
        if (lightIsOn){  
            System.out.println("The light is ON with intensity " + intensity);  
        }else{  
            System.out.println("The light is OFF");  
        }  
    }  
}
```

Οι μεταβλητές consumption είναι τοπικές μεταβλητές

Υπάρχουν μόνο μέσα στις μεθόδους dim και brighten και όταν τελειώσει η κλήση τους εξαφανίζονται.

Τοπικές μεταβλητές

- Είδαμε πρώτη φορά τις **τοπικές μεταβλητές** όταν μιλήσαμε για μεταβλητές που ορίζονται μέσα σε ένα λογικό block.
 - Παρόμοια είναι και για τις μεταβλητές μιας **μεθόδου**.
- Τοπικές μεταβλητές μιας μεθόδου είναι οι μεταβλητές που ορίζονται **μέσα** στον κώδικα της μεθόδου
 - Περιλαμβάνουν και τις μεταβλητές που κρατάνε τις **παραμέτρους** της μεθόδου
- Οι μεταβλητές αυτές έχουν **εμβέλεια** μόνο **μέσα στην μέθοδο**
 - **Εξαφανίζονται** όταν **βγούμε** από τη μέθοδο.
- Αντιθέτως τα **πεδία** της κλάσης διατηρούνται όσο υπάρχει το **αντικείμενο**, και έχουν εμβέλεια σε **όλη** την κλάση

Παράδειγμα

- Θέλουμε ένα πρόγραμμα που να προσομοιώνει την κίνηση ενός αυτοκινήτου, το οποίο κινείται και τυπώνει τη θέση του.

MovingCar

```
class Car
{
    private int position = 0;

    public void move(){
        position += 1;
    }

    public void printPosition(){
        System.out.println("Car at position "+position);
    }
}

class MovingCar
{
    public static void main(String args[]){
        Car myCar = new Car();
        myCar.move();
        myCar.printPosition();
    }
}
```

Μέθοδοι

- Οι μέθοδοι που έχουμε δει μέχρι τώρα είναι πολύ απλές
 - Δεν έχουν **παραμέτρους** (δεν παίρνουν **ορίσματα**)
 - Δεν **επιστρέφουν τιμή**

void: δεν επιστρέφει τιμή

Δεν παίρνει ορίσματα

```
public void move()  
{  
    position += 1;  
}
```


Παράδειγμα 2

- Εκτός από την κίνηση κατά μία θέση θέλουμε να μπορούμε να κινούμε το όχημα όσες θέσεις θέλουμε είτε προς τα δεξιά (+) είτε προς τα αριστερά (-).

Παράμετροι

- Οι μέθοδοι μπορούν να έχουν **παραμέτρους**
 - Μας επιτρέπουν να περάσουμε **τιμές** στην μέθοδο μας

```
public void moveManySteps(int steps)  
{  
    position += steps;  
}
```

Ορισμός
παραμέτρου

- Μία **πaráμετρος** ορίζεται όπως οποιαδήποτε άλλη **μεταβλητή**.
 - Πρέπει να έχει συγκεκριμένο **τύπο** και **όνομα**
 - Είναι **τοπική μεταβλητή** της μεθόδου

```
int x = 10;  
myCar.moveManySteps(x);  
myCar.moveManySteps(10);
```

Όρισμα στην κλήση
της μεθόδου

- Όταν καλούμε την μέθοδο, περνάμε το **όρισμα**
 - Το όρισμα είναι μια **έκφραση** (κάτι που θα μπορούσε να είναι στο δεξί μέρος μιας ανάθεσης)
 - Θα πρέπει να **συμφωνεί στον τύπο** με την παράμετρο
 - Είναι σαν να κάνουμε ανάθεση **steps = x** ή **steps = 10**

```

class Car
{
    private int position = 0;

    public void moveManySteps(int steps)
    {
        position += steps;
    }
}

class MovingCar2
{
    public static void main(String args[])
    {
        Car myCar = new Car();
        int x = 10;
        myCar.moveManySteps(x);
        myCar.moveManySteps(10);
        myCar.moveManySteps(2*x+10);
    }
}

```

Στον ορισμό της μεθόδου ορίζουμε και την **παράμετρο** της μεθόδου, όπως ορίζουμε μια μεταβλητή. Έχει ένα **τύπο** και ένα **όνομα**

Όταν καλούμε την μέθοδο περνάμε μια τιμή σαν **όρισμα** στην μέθοδο. Σαν όρισμα μπορεί να είναι μια οποιαδήποτε **έκφραση**. Αρκεί ή αποτίμηση της έκφρασης να έχει τύπο **συμβατό** με αυτόν της παραμέτρου (int στην περίπτωση μας)

Κατά την κλήση της μεθόδου ουσιαστικά **εκχωρείται** η τιμή της έκφρασης στην μεταβλητή steps. Αυτό λέγεται και **πέραςμα παραμέτρου**.

Πέρασμα παραμέτρων

- Όταν καλούμε μια μέθοδο με μία τιμή σαν όρισμα, ουσιαστικά εκχωρούμε αυτή την τιμή στην παράμετρο της μεθόδου

Η κλήση

```
myCar.moveManySteps(2*x+10);
```

όπου η μεταβλητή x έχει την τιμή 10

Αποτιμάται η τιμή της έκφρασης και εκχωρείται

Ισοδυναμεί με τον κώδικα:

```
{  
    int steps = 30;  
    position += delta;  
}
```

Το πέρασμα μεταβλητών με αυτό τον τρόπο λέγεται πέρασμα **δια τιμής (pass by value)**. Η μέθοδος δεν έχει πρόσβαση στην μεταβλητή μόνο στην τιμή

Πέρασμα παραμέτρων δια τιμής

- Όταν το πέρασμα παραμέτρων γίνεται δια τιμής, το πρόγραμμα μας έχει πρόσβαση μόνο στην τιμή της παραμέτρου και όχι στην μεταβλητή που χρησιμοποιήσαμε στο όρισμα.
 - Σε όλες τις γλώσσες πλέον το πέρασμα παραμέτρων γίνεται δια τιμής
- Αν η παράμετρος είναι ένα αντικείμενο τα πράγματα γίνονται πιο σύνθετα
 - Η τιμή της μεταβλητής που έχουμε σαν παράμετρο είναι διεύθυνση μνήμης. Δεν μπορούμε να αλλάξουμε την διεύθυνση μνήμης αλλά μπορούμε να αλλάξουμε τα περιεχόμενα της.

```
class Car
```

```
{
```

```
    private int position = 0;
```

Μέθοδος με πολλές παραμέτρους

```
    public void moveManySteps(int steps, String direction)
```

```
    {
```

```
        if (direction.equals("right") { position += steps;}
```

```
        if (direction.equals("left") { position -= steps;}
```

```
    }
```

```
}
```

Τα ορίσματα θα πρέπει να **συμφωνούν** με το **πλήθος** και τους **τύπους** των παραμέτρων στην αντίστοιχη θέση

```
class MovingCar3
```

```
{
```

```
    public static void main(String args[]) {
```

```
        Car myCar = new Car();
```

```
        myCar.moveManySteps(10, "left");
```

Κλήση της μεθόδου

```
    }
```

```
}
```

Τύποι παραμέτρων και ορισμάτων

- Οι παράμετροι μιας μεθόδου έχουν συγκεκριμένο **τύπο**
- Τα **ορίσματα** στην **κλήση** της μεθόδου θα πρέπει να **συμφωνούν με τον τύπο της παραμέτρου**, **θέση προς θέση**.
- Ισχύουν οι μετατροπές τύπου που ξέρουμε
 - **byte** → **short** → **int** → **long** → **float** → **double**
- Μία μέθοδος μπορεί να πάρει ως όρισμα και ένα **αντικείμενο** μιας κλάσης.
 - Το πώς δουλεύει αυτό θα το μάθουμε όταν μιλήσουμε για αναφορές.

Μέθοδοι που επιστρέφουν τιμές

- Μέχρι τώρα οι μέθοδοι που φτιάξαμε δεν επιστρέφουν τιμή
 - Είναι τύπου `void`.
- Σε πολλές περιπτώσεις θέλουμε η μέθοδος να μας **επιστρέφει τιμή**
 - Π.χ., μία μέθοδος που υπολογίζει το άθροισμα δύο αριθμών

Η εντολή return

- Η εντολή **return** χρησιμοποιείται για να επιστρέψει μια τιμή μια μέθοδος.
- ΣΥΝΤΑΚΤΙΚΟ:
 - **return** <έκφραση>
- Ο **τύπος** της **έκφρασης** στην εντολή **return** θα πρέπει να είναι ίδιος (ή συμβατός) με τον **τύπο** της **μεθόδου**.
- **Κάθε μονοπάτι** εκτέλεσης του κώδικα θα πρέπει να επιστρέφει μια τιμή.
- Η κλήση της **return** σε οποιοδήποτε σημείο του κώδικα **σταματάει την εκτέλεση** της μεθόδου και επιστρέφει τιμή.
 - Μπορούμε να το χρησιμοποιήσουμε αυτό για να απλοποιήσουμε τον κώδικα.

Παράδειγμα 3

- Το αυτοκίνητο μας δεν μπορεί να μετακινηθεί έξω από το διάστημα $[-10, 10]$. Θέλουμε η `moveManySteps` να μας **επιστρέφει** μια λογική τιμή αν η μετακίνηση έγινε η όχι.

Όταν ορίζουμε μια μέθοδο που επιστρέφει τιμή θα πρέπει να ορίσουμε τον **τύπο** της τιμής που επιστρέφει.

Π.χ. αυτή η μέθοδος επιστρέφει τιμή boolean

Μια μέθοδος μπορεί να επιστρέφει και ένα αντικείμενο μιας κλάσης

```
class Car
{
    private int position = 0;
```

```
public boolean moveManySteps(int steps)
```

```
{
    if ((position + steps < -10) || (position + steps > 10)) {
        return false;
    } else {
        position += steps;
        return true;
    }
}
```

Επιστρέφουμε μια τιμή μέσα στον κώδικα χρησιμοποιώντας την εντολή **return**.

```
class Car
{
    private int position = 0;

    public boolean moveManySteps(int steps)
    {
        if ((position + steps < -10) || (position + steps > 10)) {
            return false;
        }
        position += steps;
        return true;
    }
}
```

Αν μπούμε μέσα στο if η return θα σταματήσει την εκτέλεση του κώδικα και θα μας βγάλει από την μέθοδο. Επιστρέφεται η τιμή false.

Δεν χρειάζεται πλέον το else

Ο τύπος μιας μεθόδου

- Μια μέθοδος που επιστρέφει τιμή ορίζεται με συγκεκριμένο τύπο. Π.χ.
 - `public boolean moveManySteps(int steps)`
 - `public double division(int x, int y)`
 - `public String getUsername()`
 - `public Car getCar()`
- Αν έχουμε μια συνάρτηση που επιστρέφει τιμή τύπου **T**
 - Π.χ. `public double division(int x, int y)`
η έκφραση στο `return` πρέπει να επιστρέφει μία τιμή τύπου (συμβατού με το) **T**. (π.χ., `return x / (double) y`)

```
import java.util.Scanner;
```

```
class Car
```

```
{
```

```
    private int position = 0;
```

```
    public boolean moveManySteps(int steps){
```

```
        if ((position + steps < -10) || (position + steps > 10)){
```

```
            return false;
```

```
        }
```

```
        position += steps;
```

```
        return true;
```

```
    }
```

```
    public void printPosition(){
```

```
        System.out.println("Car at position "+position);
```

```
    }
```

```
}
```

```
class MovingCar4b{
```

```
    public static void main(String args[]){
```

```
        Scanner input = new Scanner(System.in);
```

```
        Car myCar = new Car();
```

```
        int steps = input.nextInt();
```

```
        boolean carMoved = myCar.moveManySteps(steps);
```

```
        if (carMoved) { myCar.printPosition();}
```

```
        else { System.out.println("Car could not move");}
```

```
    }
```

```
}
```

Κλήση της μεθόδου

```
import java.util.Scanner;
```

```
class Car
```

```
{
```

```
    private int position = 0;
```

```
    public boolean moveManySteps(int steps){
```

```
        if ((position + steps < -10) || (position + steps > 10)){
```

```
            return false;
```

```
        }
```

```
        position += steps;
```

```
        return true;
```

```
    }
```

```
    public void printPosition(){
```

```
        System.out.println("Car at position "+position);
```

```
    }
```

```
}
```

```
class MovingCar4b{
```

```
    public static void main(String args[]){
```

```
        Scanner input = new Scanner(System.in);
```

```
        Car myCar = new Car();
```

```
        int steps = input.nextInt();
```

```
        if (myCar.moveManySteps(steps) ) {
```

```
            myCar.printPosition();
```

```
        }
```

```
        else { System.out.println("Car could not move");}
```

```
    }
```

```
}
```

Κλήση της μεθόδου και χρήση του αποτελέσματος απευθείας μέσα στην συνθήκη. Δεν χρειάζεται να το αποθηκεύσουμε.

```
import java.util.Scanner;
```

```
class Car
{
    private int position = 0;

    public boolean moveManySteps(int steps)
    {
        if ((position + steps < -10) || (position + steps > 10)){
            return false;
        }
        position += steps;
        return true;
    }
    public void printPosition(){
        System.out.println("Car at position "+position);
    }
}
```

```
class MovingCar4c
{
    public static void main(String args[]){
        Scanner input = new Scanner(System.in);
        Car myCar = new Car();
        int steps = input.nextInt();
        myCar.moveManySteps(steps);
        myCar.printPosition();
    }
}
```

Δεν είναι υποχρεωτικό να χρησιμοποιούμε πάντα την επιστρεφόμενη τιμή

Η moveManySteps επιστρέφει τιμή, αλλά η κλήση της την αγνοεί

Η printPosition θα επιστρέψει 0 αν δεν κινήθηκε το όχημα

Η εντολή return

- Μπορούμε να καλέσουμε την **return** και σε μία **void** μέθοδο
 - Χωρίς επιστρεφόμενη τιμή.
 - **return;**
 - Σταματάει την εκτέλεση της μεθόδου

```
public void printIfPositive()  
{  
    if (position < 0) {  
        return;  
    }  
    System.out.println("position = " + position);  
}
```

Η εντολή return

- Μπορούμε να καλέσουμε την **return** και σε μία **void** μέθοδο
 - Χωρίς επιστρεφόμενη τιμή.
 - **return;**
 - Σταματάει την εκτέλεση της μεθόδου

```
public void moveManySteps(int steps, String direction)
{
    if (steps < 0) {
        return;
    }
    if (direction.equals("right") { position += steps;}
    if (direction.equals("left") { position -= steps;}
}
```

Παράδειγμα 4

- Θέλουμε να μπορούμε να κινούμε το όχημα όσες θέσεις θέλουμε είτε προς τα δεξιά (+) είτε προς τα αριστερά (-), **και** να τυπώνεται η θέση σε κάθε κίνηση.
- Υλοποίηση: Θα ορίσουμε μια βοηθητική μεταβλητή `delta` την οποία θα προσθέτουμε στο `position` σε κάθε βήμα. Η default τιμή του θα είναι `delta = 1`. Αν η παράμετρος `steps` είναι αρνητική θα την μετατρέπουμε σε θετική και θα θέσουμε `delta = -1`.

```

class Car
{
    private int position = 0;

    public void moveManySteps(int steps)
    {
        int delta = 1;
        if (steps < 0){
            steps = -steps; delta = -1;
        }
        for (int i = 0; i < steps; i ++){
            position += delta;
            System.out.println("Car at position "+position);
        }
    }
    public void printPosition(){
        System.out.println("Car at position "+position);
    }
}

class MovingCar5
{
    public static void main(String args[]){
        Car myCar = new Car();
        int steps = -10;
        myCar.moveManySteps(steps);
        System.out.println("--: " + steps);
    }
}

```

Το **delta** είναι **τοπική μεταβλητή** της μεθόδου. Ορίζεται μέσα στην μέθοδο και υπάρχει μόνο μέσα στην μέθοδο. Στο τέλος της μεθόδου η μεταβλητή χάνεται.

Η **παράμετρος steps** λειτουργεί ως **τοπική μεταβλητή** της συνάρτησης και χάνεται μετά την κλήση της μεθόδου.

Η **μεταβλητή steps** στην main είναι διαφορετική από την **παράμετρο steps** στην μέθοδο. Το πέρασμα παραμέτρων γίνεται δια τιμής και άρα η τιμή της μεταβλητής του **ορίσματος δεν** μεταβάλλεται

Τυπώνει --: -10

```
class Car
{
    private int position = 0;

    public void moveManySteps(int steps)
    {
        int delta = 1;
        if (steps < 0){
            steps = -steps; delta = -1;
        }
        for (int i = 0; i < steps; i ++){
            position += delta;
            printPosition();
        }
    }

    public void printPosition(){
        System.out.println("Car at position "+position);
    }
}
```

Μπορούμε να κάνουμε την εκτύπωση καλώντας την printPosition()

Κάθε μέθοδος που ορίζουμε μέσα σε μία κλάση μπορούμε να την χρησιμοποιήσουμε και μέσα στην κλάση

Παράδειγμα 4

- Όταν καλούμε την συνάρτηση `move()` το όχημα μας θα κινείται ένα **τυχαίο αριθμό** από βήματα στο διάστημα $(-3,3)$

Υλοποίηση

- Θα φτιάξουμε μια **βοηθητική συνάρτηση** που θα μας **επιστρέφει** τον τυχαίο αριθμό από βήματα.

private: δεν χρειάζεται να φαίνεται έξω από την κλάση

```
private int computeRandomSteps ()
{
    int radomSteps;
    // do the computation

    return randomSteps;
}

public void move () {
    int steps = computeRandomSteps ();
    moveManySteps (steps);
}
```

Κλήση της
συνάρτησης και
χρήση της
επιστρεφόμενης
τιμής

```
import java.util.Random;
```

```
class Car
{
    private int MAX_VALUE = 3;
    private int position = 0;
    private Random randomGenerator = new Random();

    private int computeRandomSteps ()
    {
        int randomSteps = randomGenerator.nextInt(2*MAX_VALUE + 1) - MAX_VALUE;
        return randomSteps;
    }

    public void move() {
        int steps = computeRandomSteps ();
        moveManySteps (steps);
    }

    public void moveManySteps(int steps) { ... }

    public void printPosition() {
        System.out.println("Car at position "+position);
    }
}

class MovingCar6
{
    public static void main(String args[]){
        Car myCar = new Car();
        myCar.move();
    }
}
```

Η κλάση **Random**: Δημιουργεί μια γεννήτρια τυχαίων αριθμών που παράγει τυχαίους αριθμούς

Μέθοδος **nextInt(int x)** της Random: Επιστρέφει ένα τυχαίο ακέραιο αριθμό στο διάστημα [0, x)

Public/Private

- Ότι είναι ορισμένο ως **public** σε μία κλάση είναι προσβάσιμο από **οποιονδήποτε**.
 - Μπορούμε να καλέσουμε τις μεθόδους ορίζοντας ένα αντικείμενο της κλάσης
- Ότι είναι ορισμένο ως **private** σε μία κλάση είναι προσβάσιμο **μόνο** από την **ίδια κλάση**.
- Ο τροποποιητής **private** μας επιτρέπει την **απόκρυψη πληροφοριών** (**information hiding**).
 - Ο χρήστης της κλάσης **Car**, δεν χρειάζεται να ξέρει πως υλοποιείται η μέθοδος **computeRandomSteps** που υπολογίζει τον τυχαίο αριθμό των βημάτων.
 - Αν αποφασίσουμε να αλλάξουμε κάτι στη μέθοδο αυτό θα γίνει ως μέρος του επανασχεδιασμού της κλάσης **Car**. Κανείς άλλος δεν θα πρέπει να επηρεαστεί από την αλλαγή στον κώδικα.
- Τα **πεδία** μιας κλάσης τα ορίζουμε **πάντα private**.

Ενθυλάκωση

- Η ομαδοποίηση λογισμικού και δεδομένων σε μία οντότητα (κλάση και αντικείμενα της κλάσης) ώστε να είναι εύχρηστη μέσω ενός καλά ορισμένου **interface**, ενώ οι λεπτομέρειες υλοποίησης είναι κρυμμένες από τον χρήστη.
- **API** (Application Programming Interface)[‘Ει-Πι-Άι]
 - Μια περιγραφή για το πώς χρησιμοποιείται η κλάση μέσω των **public μεθόδων** της.
 - Java docs είναι ένα παράδειγμα.
 - Το API είναι αρκετό για να χρησιμοποιήσετε μια κλάση, δεν χρειάζεται να ξέρετε την υλοποίηση των μεθόδων.
- **ADT** (Abstract Data Type)
 - Ένας τύπος δεδομένων που ορίζεται χρησιμοποιώντας την αρχή της ενθυλάκωσης
 - Οι λίστες που χρησιμοποιήσατε στην Python είναι ένα παράδειγμα.
 - Δεδομένα και μέθοδοι.

An encapsulated class

Implementation details hidden in the capsule:

Private instance variables
Private constants
Private methods
Bodies of public and private method definitions

Interface available to a programmer using the class:

Comments
Headings of public accessor, mutator, and other methods
Public defined constants

Programmer who uses the class

A class definition should have no public instance variables.

Accessor and Mutator methods

- Πολλές φορές χρειαζόμαστε να **διαβάσουμε** ή να **αλλάξουμε** ένα πεδίο ενός αντικειμένου
 - Π.χ., να διαβάσουμε τη θέση του οχήματος, ή να τοποθετήσουμε το όχημα σε μια συγκεκριμένη θέση.
 - Πως θα το κάνουμε αφού τα πεδία είναι private?
- Ορίζουμε ειδικές μεθόδους
 - **Μέθοδος προσπέλασης** (**accessor** method) για διάβασμα
 - **Μέθοδος μεταλλαγής** (**mutator** method) για γράψιμο
- **Σύμβαση**: Στη Java η ονοματολογία των μεθόδων αυτών γίνεται με συγκεκριμένο τρόπο:
 - **get<ονομα μεταβλητης>** για την πρόσβαση
 - getPosition()
 - **set<ονομα μεταβλητης>** για την μετάλλαξη
 - setPosition(<τιμή>)

```
class Car
{
    private int position = 0;

    public void setPosition(int p){
        position = p;
    }

    public int getPosition(){
        return position;
    }

    public void move(){
        position ++ ;
    }
}

class MovingCar7
{
    public static void main(String args[]){
        Car myCar = new Car();
        myCar.setPosition(10);
        myCar.move();
        System.out.println(myCar.getPosition());
    }
}
```

Υπάρχουν περιπτώσεις που μπορεί να θέλουμε η συνάρτηση set να επιστρέφει **boolean** (true αν η ανάθεση έγινε επιτυχώς, false αλλιώς)

```
class Car
{
    private int position = 0;

    public boolean setPosition(int p){
        if (p < 0){
            return false;
        }
        position = p;
        return true;
    }
}
```

```
    public int getPosition(){
        return position;
    }

    public void move(){
        position ++ ;
    }
}
```

```
class MovingCar7b
{
    public static void main(String args[]){
        Car myCar = new Car();
        boolean check = myCar.setPosition(-1);
        if (!check){
            System.out.println("position not set");
        }
    }
}
```

Η setPosition μπορεί να επιστρέφει τιμή
Το πιο συνηθισμένο είναι να επιστρέφει
boolean αν έγινε σωστά η ανάθεση

Τοπικές μεταβλητές

- Οι τοπικές μεταβλητές (και οι παράμετροι) που ορίζουμε μέσα σε μία μέθοδο, έχουν προτεραιότητα σε σχέση με τα πεδία της μεθόδου
 - Δηλαδή αν έχουμε μια τοπική μεταβλητή με το ίδιο όνομα με ένα πεδίο μέσα σε μία μέθοδο, όταν χρησιμοποιούμε το όνομα αναφερόμαστε στην τοπική μεταβλητή και όχι στο πεδίο.
 - Αν θέλουμε να αναφερθούμε στο πεδίο μπορούμε να χρησιμοποιήσουμε την δεσμευμένη λέξη **this**.

Το αντικείμενο this

- Με την δεσμευμένη λέξη **this** ένα αντικείμενο αναφέρεται στον εαυτό του.

```
class Car
{
    private int position = 0;

    public void setPositionToTen() {
        this.position = 10;
    }
}
```

```
class Example
{
    public static void main(String args[]) {
        Car myCar = new Car();
        myCar.setPositionToTen();
    }
}
```

Το **this.position** αναφέρεται στο εσωτερικό πεδίο position του αντικείμενου που θα καλέσει την μέθοδο.

Όταν δημιουργηθεί το αντικείμενο myCar και καλέσει την μέθοδο setPositionToTen το **this** αναφέρεται πλέον στο αντικείμενο που κάλεσε την μέθοδο, δηλαδή το myCar


```
class Car
{
    private int position = 0;

    public void setPosition(int position) {
        this.position = position;
    }

    public int getPosition() {
        return position;
    }

    public void move() {
        position ++ ;
    }
}

class MovingCar8
{
    public static void main(String args[]) {
        Car myCar = new Car();
        myCar.setPosition(10);
        myCar.move();
        System.out.println(myCar.getPosition());
    }
}
```

Το **this.position** αναφέρεται στο πεδίο του αντικειμένου.
Το **position** αναφέρεται στην παράμετρο της συνάρτησης

Το κρυφό πεδίο **this** προσδιορίζει το αντικείμενο που κάλεσε την μέθοδο

Έτσι μπορούμε να χρησιμοποιήσουμε το ίδιο όνομα μεταβλητής χωρίς να δημιουργείται σύγχυση

```
class LocalVariableTest
```

```
{
```

```
private int var = 10;
```

Ορισμός του πεδίου var

```
public void method1() {
```

```
int var = 5;
```

```
var ++;
```

Ορισμός τοπικής μεταβλητής var.
Η χρήση της var μέσα στην μέθοδο αναφέρεται στην τοπική μεταβλητή

```
}
```

```
public void method2(int var) {
```

```
var ++;
```

Ορισμός παραμέτρου var.
Η χρήση της var μέσα στην μέθοδο αναφέρεται στην παράμετρο

```
}
```

```
public void method3() {
```

```
int var = 1;
```

```
this.var = var;
```

Ορισμός τοπικής μεταβλητής var.
Η χρήση της var μέσα στην μέθοδο αναφέρεται στην τοπική μεταβλητή.
Το this.var αναφέρεται στο πεδίο της κλάσης

```
}
```

```
public void printVar() {
```

```
System.out.println("var = "+var);
```

Δεν υπάρχει τοπική μεταβλητή,
χρήση του πεδίου var

```
}
```

```
public static void main(String[] args) {
```

```
LocalVariableTest x = new LocalVariableTest();
```

```
int var = 3;
```

```
x.method1(); x.printVar();
```

```
x.method2(var); x.printVar();
```

```
x.method3(); x.printVar();
```

```
System.out.println(var);
```

```
}
```

```
}
```

Τι θα τυπώσει?

var = 10

var = 10

var = 1

var = 3

Ορισμός και χρήση
τοπικής μεταβλητής
var στην main.

Παρένθεση: Μπορούμε να ορίσουμε main
μέσα σε μία κλάση για να την τεστάρουμε

Constructors (Δημιουργοί)

- Ο **Constructor** είναι μια «μέθοδος» η οποία καλείται όταν **δημιουργούμε** το αντικείμενο χρησιμοποιώντας την **new**.
- Αν δεν έχουμε ορίσει Constructor καλείται ένας **default Constructor** χωρίς ορίσματα που δεν κάνει τίποτα.
 - Ο default constructor απλά εκτελεί τις αρχικοποιήσεις.
- Αν ορίσουμε constructor, τότε καλείται ο constructor που **ορίσαμε**.

ΣΥΝΤΑΚΤΙΚΟ

- Ο constructor είναι μια μέθοδος:
 - Που έχει το όνομα της κλάσης
 - Ορίζεται πάντα **public**
 - Δεν έχει **τύπο**

```
class <Όνομα κλάσης>
{
    <Ορισμός Πεδίων>
    public <Όνομα κλάσης> ( [ορίσματα] )
    {
        [κώδικας] ;
    }
}
```

Παράδειγμα

```
class Person
{
    private String name;

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

    public void speak(String s) {
        System.out.println(name+": "+s);
    }
}

public class HelloWorld2
{
    public static void main(String[] args) {
        Person alice = new Person("Alice");
        alice.speak("Hello World");
    }
}
```

Constructor: μια μέθοδος με το ίδιο όνομα όπως και η κλάση και **χωρίς τύπο** (ούτε void)

Αρχικοποιεί την μεταβλητή name

Constructor: καλείται όταν δημιουργείται το αντικείμενο με την **new** και **μόνο** τότε

Μια συνομιλία

```
class Person
{
    private String name;

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

    public void speak(String s){
        System.out.println(name+": "+s);
    }
}

public class Conversation
{
    public static void main(String[] args){
        Person alice = new Person("Alice");
        Person bob = new Person("Bob");
        alice.speak("Hi Bob");
        bob.speak("Hi Alice");
    }
}
```

Παράδειγμα

```
class Car
{
    private int position;

    public Car(int position){
        this.position = position;
    }

    public void move(int delta){
        position += delta ;
    }

    public void printPosition(){
        System.out.println("Car is at position "+position);
    }
}

class MovingCar9
{
    public static void main(String args[]){
        Car myCar1 = new Car(1);
        Car myCar2 = new Car(-1);
        myCar1.move(-1); myCar1.printPosition();
        myCar2.move(1); myCar2.printPosition();
    }
}
```

Παράδειγμα

```
class Car
{
    private int position=0;
    private int ACCELERATOR = 2;

    public Car(int position){
        this.position = position;
    }

    public void move(int delta){
        position += ACCELERATOR * delta ;
    }

    public void printPosition(){
        System.out.println("Car is at position "+position);
    }
}

class MovingCar10
{
    public static void main(String args[]){
        Car myCar1 = new Car(1);
        Car myCar2 = new Car(-1);
        myCar1.move(-1); myCar1. printPosition();
        myCar2.move(1); myCar2. printPosition();
    }
}
```

Η εκτέλεση αυτών των
αρχικοποιήσεων γίνεται
πριν εκτελεστούν οι
εντολές στον constructor

Η τελική τιμή του position θα είναι
αυτή που δίνεται σαν όρισμα

Παράδειγμα

- Μία κλάση που να αποθηκεύει ημερομηνίες
 - Η κλάση θα παίρνει την ημέρα, μήνα και χρόνο σαν νούμερα (π.χ., 13 3 2014) και θα μπορεί να τυπώνει την ημερομηνία με το όνομα του μήνα (π.χ., 13 Μαρτίου 2014)
 - Στο πρόγραμμα βάλετε μια ημερομηνία και τυπώστε την.

```
class Date
{
    private int day = 1;
    private int month = 1;
    private int year = 2016;
    private String[] monthNames = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    public Date(int day, int month, int year)
    {
        if (day <= 0 || day > 31 || month <= 0 || month >12 ){
            return;
        }
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public void printDate(){
        System.out.println(day + " " + monthNames[month-1] + " " + year);
    }
}

class DateExample
{
    public static void main(String args[])
    {
        Date myDate = new Date(9,3,2016);
        myDate.printDate();
    }
}
```

```

class Date
{
    private int day; private int month; private int year;
    private String[] monthNames =
        {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    public Date(int day, int month, int year)
    {
        if (checkDay(day)) { this.day = day; }
        if (checkMonth(month)) { this.month = month; }
        this.year = year;
    }

    private boolean checkDay(int day) {
        if (day <= 0 || day > 31 ) {return false;}
        return true;
    }

    private boolean checkMonth(int day) {
        if (month <= 0 || month >12) {return false;}
        return true;
    }

    public void printDate()
    {
        System.out.println(day + " " + monthNames[month-1] + " " + year);
    }
}

```

Ο constructor μπορεί να καλεί και άλλες μεθόδους που κάνουν κάποια από τη δουλειά που χρειάζεται

Η εκτέλεση αυτών των αρχικοποιήσεων γίνεται **πριν** εκτελεστούν οι εντολές στον constructor

```
class Date
{
    private int day = 1;
    private int month = 1;
    private int year = 2016;
    private String[] monthNames = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    public Date(int day, int month, int year)
    {
        if (day <= 0 || day > 31 || month <= 0 || month >12 ){
            return;
        }
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public void printDate(){
        System.out.println(day + " " + monthNames[month-1] + " " + year);
    }
}

class DateExample
{
    public static void main(String args[])
    {
        Date myDate = new Date(9,3,2016);
        myDate.printDate();
    }
}
```

Αν μπούμε στο if οι τελικές τιμές των ορισμάτων θα είναι αυτές που θα δοθούν στον constructor . Αλλιώς διατηρούνται οι αρχικές τιμές

Παραδείγματα

- Θέλουμε μια κλάση **Student** που να κρατάει πληροφορίες για έναν φοιτητή. Τι πεδία πρέπει να έχουμε? Τι θα μπει στον constructor?
- Θέλουμε μια κλάση (**GuestList**) που να χειρίζεται τους καλεσμένους σε ένα πάρτι. Τι πεδία πρέπει να έχουμε? Πώς θα κάνουμε τον constructor?

```
class Student
{
    private String name = "John Doe";
    private int AM = 1000;

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

    public void printInfo() {
        System.out.println(name + " " + AM);
    }

    public static void main(String[] args) {
        Student aStudent = new Student("Kostas", 1001);
        aStudent.printInfo();
    }
}
```

Guest List

Ορίζει τους πίνακες με τα ονόματα των καλεσμένων και τις επιβεβαιώσεις, αλλά δεν δεσμεύει χώρο γιατί δεν ξέρουμε τον αριθμό των προσκεκλημένων.

Δεσμεύει μνήμη για τους πίνακες με τα ονόματα των καλεσμένων και τις επιβεβαιώσεις

```
class GuestList
```

```
{  
    private String[] names;  
    private boolean[] confirm;  
    int numberOfGuests;  
  
    public GuestList(int numberOfGuests)  
    {
```

Αρχικοποίηση του αριθμού των επισκεπτών

```
        this.numberOfGuests = numberOfGuests;
```

```
        names = new String[numberOfGuests];  
        confirm = new boolean[numberOfGuests];
```

```
        // Εδώ μπορούμε να έχουμε κώδικα για τις τιμές  
        // ή να εισάγονται τα ονόματα ένα ένα.  
        // ή μπορεί η εισαγωγή ονομάτων να γίνει  
        // με άλλες μεθόδους της κλάσης
```

```
    }
```

```
}
```

Μια υλοποίηση με μέθοδο για
προσθήκη προσκεκλημένων

```
class GuestList
```

```
{
```

```
    private String[] names;
```

```
    private boolean[] confirm;
```

```
    int numberOfGuests;
```

```
    int guestsSoFar = 0;
```

Χρειαζόμαστε αυτή τη μεταβλητή
για να ξέρουμε πόσους επισκέπτες
έχουμε προσθέσει μέχρι τώρα

```
    public GuestList(int numberOfGuests)
```

```
    {
```

```
        this.numberOfGuests = numberOfGuests;
```

```
        names = new String[numberOfGuests];
```

```
        confirm = new boolean[numberOfGuests];
```

```
    }
```

```
    public void addGuest(String name, Boolean confirmation) {
```

```
        if (guestsSoFar == numberOfGuests) {
```

```
            return;
```

```
        }
```

```
        names[guestsSoFar] = name;
```

```
        confirm[guestsSoFar] = confirmation;
```

```
        guestsSoFar ++;
```

```
    }
```

```
}
```

Αν έχει γεμίσει η
λίστα μας δεν
προσθέτουμε

Η guestsSoFar μας δίνει
και την επόμενη άδεια
θέση στον πίνακα.

5. ΔΗΜΙΟΥΡΓΙΑ ΚΛΑΣΕΩΝ ΚΑΙ ΑΝΤΙΚΕΙΜΕΝΩΝ II

Υπερφόρτωση

Αντικείμενα ως παράμετροι

Οι μέθοδοι equals και toString

Αντικείμενα με αντικείμενα

Αντικείμενα ως επιστρεφόμενες τιμές

Αντικείμενα με πίνακες

ΥΠΕΡΦΟΡΤΩΣΗ

Η κλάση Car

- Μια κλάση που κρατάει την θέση ενός αυτοκινήτου.
 - Μέθοδος `move()`: μετακινεί το αυτοκίνητο κατά μία θέση προς τα δεξιά
 - Μέθοδος `moveManySteps(int delta)`: μετακινεί `delta` θέσεις (αρνητικά ή θετικά).
- Και οι δύο μέθοδοι ουσιαστικά υλοποιούν το `move` απλά η μία παίρνει όρισμα και η άλλη όχι.
 - Θα ήταν καλύτερα να μπορούσαμε να χρησιμοποιήσουμε το ίδιο όνομα και για τις δύο συναρτήσεις.
- Η Java μας δίνει αυτή τη δυνατότητα μέσω της διαδικασίας της **υπερφόρτωσης (overloading)**

Υπερφόρτωση (Overloading)

- Η Java μας δίνει τη δυνατότητα να ορίσουμε την πολλές μεθόδους με το ίδιο όνομα μέσω της διαδικασίας της **υπερφόρτωσης (overloading)**
 - Ορισμός πολλών μεθόδων με το **ίδιο όνομα** αλλά **διαφορετικά ορίσματα**, μέσα στην ίδια κλάση.
- Για να μπορεί να γίνει σωστά η υπερφόρτωση θα πρέπει οι μέθοδοι να έχουν διαφορετική **υπογραφή**
- Η **υπογραφή** μίας μεθόδου είναι το **όνομα** της και η **λίστα με τους τύπους των ορισμάτων** της μεθόδου
 - Η Java μπορεί να ξεχωρίσει μεθόδους με διαφορετική υπογραφή.

```
class Car
{
    private int position;

    public Car(int position){
        this.position = position;
    }

    public void move() {
        position ++ ;
    }

    public void move(int delta) {
        position += delta ;
    }
}
```

Οι μέθοδοι `move()` και `move(int)` έχουν διαφορετική υπογραφή

```
class MovingCar11
{
    public static void main(String args[]) {
        Car myCar = new Car(1);
        myCar.move() ;
        myCar.move(-1) ;
    }
}
```

Μετακινεί το όχημα μια θέση μπροστά
Μετακινεί το όχημα μια θέση πίσω

Υπερφόρτωση Δημιουργών

- Είναι αρκετά συνηθισμένο να υπερφορτώνουμε τους δημιουργούς (constructors) των κλάσεων.

Υπερφόρτωση δημιουργιών

```
class Car
{
    private int position;

    public Car(){
        this.position = 0;
    }

    public Car(int position){
        this.position = position;
    }

    public void move(){
        position ++ ;
    }

    public void move(int delta){
        position += delta ;
    }
}

class MovingCar12
{
    public static void main(String args[]){
        Car myCar1 = new Car(1); myCar1.move();
        Car myCar2= new Car(); myCar2.move(-1);
    }
}
```

```
class Car
{
    private int position = 0;

    public Car() {}

    public Car(int position) {
        this.position = position;
    }

    public void move() {
        position ++ ;
    }

    public void move(int delta) {
        position += delta ;
    }
}

class MovingCar12
{
    public static void main(String args[]) {
        Car myCar1 = new Car(1); myCar1.move();
        Car myCar2= new Car(); myCar2.move(-1);
    }
}
```

Κενός κώδικας, χρειάζεται για να οριστεί ο “default” constructor

Γενικά είναι καλό να ορίζετε και ένα constructor χωρίς ορίσματα

Υπερφόρτωση – Προσοχή I

- Όταν ορίζουμε ένα constructor, ο default constructor **παύει να υπάρχει**. Πρέπει να τον ορίσουμε μόνοι μας.

```
class Car
{
    private int position = 0;

    public Car(int position) {
        this.position = position;
    }

    public void move() {
        position ++ ;
    }

    public void move(int delta) {
        position += delta ;
    }
}

class MovingCar12
{
    public static void main(String args[]) {
        Car myCar1 = new Car(1);
        myCar1.move();
        Car myCar2 = new Car();
        myCar2.move(-1);
    }
}
```

Θα χτυπήσει **λάθος** ότι
δεν υπάρχει constructor
χωρίς ορίσματα

Υπερφόρτωση – Προσοχή II

- Η **υπερφόρτωση** γίνεται μόνο **ως προς τα ορίσματα**, **ΌΧΙ** ως προς **την επιστρεφόμενη τιμή**.
- Η **υπογραφή** μίας μεθόδου είναι το **όνομα** της και η **λίστα με τους τύπους των ορισμάτων** της μεθόδου
 - Η Java μπορεί να ξεχωρίσει μεθόδους με διαφορετική υπογραφή.
 - Π.χ., `move()`, `move(int)` έχουν διαφορετική **υπογραφή**
- Όταν δημιουργούμε μια μέθοδο θα πρέπει να δημιουργούμε μία **διαφορετική υπογραφή**.

```
class SomeClass
```

```
{
```

```
public int aMethod(int x, double y){  
    System.out.println("int double");  
    return 1;  
}
```

A

```
public double aMethod(int x, double y){  
    System.out.println("int double");  
    return 1;  
}
```

B

```
public int aMethod(double x, int y){  
    System.out.println("double int");  
    return 1;  
}
```

C

```
public double aMethod(double x, int y){  
    System.out.println("double int");  
    return 1;  
}
```

D

```
}
```

Ποιοι συνδυασμοί είναι αποδεκτοί?

A

B



A

C



A

D



B

C



B

D



C

D



Υπερφόρτωση – Προσοχή III

- Λόγω της συμβατότητας μεταξύ τύπων μια κλήση μπορεί να ταιριάζει με διάφορες μεθόδους.
- Καλείται αυτή που ταιριάζει **ακριβώς**, ή αυτή που είναι **ΠΙΟ ΚΟΝΤΑ**.
- Αν υπάρχει **ασάφεια** θα χτυπήσει ο compiler.

```
class SomeClass
{
    public int aMethod(int x, int y){
        System.out.println("int int");
        return 1;
    }

    public float aMethod(float x, float y){
        System.out.println("float float");
        return 1;
    }

    public double aMethod(double x, double y){
        System.out.println("double double");
        return 1;
    }
}
```

Τι θα τυπώσει η κλήση της μεθόδου?

```
class OverloadingExample
{
    public static void main(String args[])
    {
        SomeClass anObject = new SomeClass();
        anObject.aMethod(1,1);
    }
}
```

Τυπώνει "int int"
γιατί **ταιριάζει** ακριβώς με τις
παραμέτρους που δώσαμε

```

class SomeClass
{
    /*
    public int aMethod(int x, int y){
        System.out.println("int int");
        return 1;
    }
    */

    public float aMethod(float x, float y){
        System.out.println("float float");
        return 1;
    }

    public double aMethod(double x, double y){
        System.out.println("double double");
        return 1;
    }
}

```

Τι θα τυπώσει η κλήση της μεθόδου?

```

class OverloadingExample
{
    public static void main(String args[])
    {
        SomeClass anObject = new SomeClass();
        anObject.aMethod(1,1);
    }
}

```

Τυπώνει "float float"
γιατί είναι **πιο κοντά** ακριβώς με
τις παραμέτρους που δώσαμε

Ασάφεια

```
class SomeClass
{
    public double aMethod(int x, double y) {
        System.out.println("int double");
        return 1;
    }

    public int aMethod(double x, int y) {
        System.out.println("double int");
        return 1;
    }
}
```

Τι θα τυπώσει η κλήση της μεθόδου σε κάθε περίπτωση?

```
class OverloadingExample
{
    public static void main(String args[])
    {
        SomeClass anObject = new SomeClass();
        anObject.aMethod(1.0, 1);
        anObject.aMethod(1, 1);
    }
}
```

Τυπώνει "double int"

Ο compiler μας πετάει λάθος γιατί η κλήση είναι ασαφής (ambiguous)

ΑΝΤΙΚΕΙΜΕΝΑ ΣΑΝ ΟΡΙΣΜΑΤΑ

Αντικείμενα ως ορίσματα

- Οποιαδήποτε κλάση μπορεί να χρησιμοποιηθεί ως παράμετρος και μπορούμε να περνάμε **αντικείμενα ως ορίσματα** σε μία μέθοδο όπως οποιαδήποτε άλλη μεταβλητή

Παράδειγμα

- Ορίστε μια μέθοδο που να μας επιστρέφει την απόσταση μεταξύ δύο οχημάτων.

```

class Car
{
    private int position = 0;

    public Car(int position){
        this.position = position;
    }

    public int getPosition() { return position;}

    public void move(int delta){
        position += delta ;
    }
}

class MovingCarDistance1
{
    public static void main(String args[]){
        Car myCar1 = new Car(1);
        Car myCar2 = new Car(0);
        myCar2.move(2);
        System.out.println("Distance of Car 1 from Car 2: " + computeDistance(myCar1,myCar2));
        System.out.println("Distance of Car 2 from Car 1: " + computeDistance(myCar2,myCar1));
    }

    private static int computeDistance(Car car1, Car car2){
        return car1.getPosition() - car2.getPosition();
    }
}

```

Μια μέθοδος ή ένα πεδίο που χρησιμοποιείται σε μία static μέθοδο πρέπει να είναι επίσης static

Η μέθοδος computeDistance παίρνει σαν όρισμα δύο αντικείμενα τύπου Car

Αντικείμενα σαν ορίσματα

- Στον αντικειμενοστραφή προγραμματισμό συνήθως δεν ορίζουμε τέτοιου είδους μεθόδους. Η **κλάση** αναλαμβάνει να υλοποιεί μεθόδους που αφορούν τα αντικείμενα της
- Οπότε μέσα στην κλάση θα πρέπει να ορίσουμε μια **μέθοδο** που να μας **δίνει την απόσταση**. Πως θα το κάνουμε?
- Θα ορίσουμε μια public μέθοδο στην Car που θα παίρνει σαν **όρισμα ένα άλλο αντικείμενο Car** και θα μας επιστρέφει την απόσταση του από το αντικείμενο που κάλεσε την μέθοδο

Αντικείμενα ως ορίσματα

- Όταν τα **ορίσματα** είναι της ίδιας **κλάσης** με αυτή στην οποία ορίζεται η **μέθοδος** τότε η μέθοδος μπορεί να δει (έχει πρόσβαση) και στα **ιδιωτικά** (private) πεδία των αντικειμένων
- Αν τα ορίσματα είναι διαφορετικού τύπου τότε η μέθοδος μπορεί να καλέσει μόνο τις **public** μεθόδους.

Διάβασμα πεδίων

- Η προσπέλαση των πεδίων (για διάβασμα ή γράψιμο) γίνεται με τον ίδιο τρόπο όπως και η προσπέλαση των μεθόδων

`<όνομα αντικειμένου>.<όνομα πεδίου>`

Παράδειγμα

- Ορίστε μια μέθοδο της κλάσης `Car` που να παίρνει σαν όρισμα ένα άλλο αντικείμενο `Car` και να μας επιστρέφει την απόσταση μεταξύ δύο οχημάτων.


```
class Car
{
    private int position = 0;

    public Car(int position){
        this.position = position;
    }

    public void move(int delta){
        position += delta ;
    }

    public int distanceFrom(Car other){
        return this.position - other.position;
    }
}

class MovingCarDistance2
{
    public static void main(String args[]){
        Car myCar1 = new Car(1);
        Car myCar2 = new Car(0); myCar2.move(2);
        System.out.println("Distance of Car 1 from Car 2: " + myCar1.distanceFrom(myCar2));
        System.out.println("Distance of Car 2 from Car 1: " + myCar2.distanceFrom(myCar1));
    }
}
```

Συνήθως προτιμούμε όποια μέθοδος έχει σχέση με την κλάση να την ορίζουμε ως public μέθοδο της κλάσης. Έχουμε επιπλέον ευελιξία γιατί έχουμε πρόσβαση σε όλα τα πεδία της κλάσης

Στο σημείο αυτό διαβάζουμε τα πεδία position για το αντικείμενο this και other. Αν και το πεδίο position είναι private μπορούμε να το προσπελάσουμε γιατί είμαστε μέσα στην κλάση Car.

Μία κλάση μπορεί να προσπελάσει τα ιδιωτικά μέλη όλων των αντικειμένων της κλάσης

Διάβασμα πεδίων

- Η προσπέλαση των πεδίων (για διάβασμα ή γράψιμο) γίνεται με τον ίδιο τρόπο όπως και η προσπέλαση των μεθόδων

`<όνομα αντικειμένου>.<όνομα πεδίου>`

- Και το αντικείμενο `this` είναι μια τέτοια περίπτωση.

```
public int distanceFrom(Car other) {  
    return this.position - other.position;  
}
```



Παράδειγμα

- Ορίστε μια μέθοδο που θα παίρνει όρισμα ένα άλλο όχημα και θα βάζει το όχημα που είναι πιο πίσω στην ίδια θέση με το όχημα που είναι πιο μπροστά.

```
class Car
```

```
{  
    private int position = 0;  
  
    public Car(){}  
  
    public void move(int delta){  
        position += delta ;  
    }  
  
    public void catchUp(Car other){  
        if (this.position > other.position){  
            this.position = other.position;  
        }else{  
            other.position = this.position;  
        }  
    }  
  
    public void printPosition(){  
        System.out.println("Car is at position "+position);  
    }  
}
```

Μπορούμε όχι μόνο να διαβάσουμε αλλά και να αλλάξουμε την τιμή του πεδίου position στο αντικείμενο other.

```
class MovingCar13{  
    public static void main(String args[]){  
        Car myCar1 = new Car(); myCar1.move(10);  
        Car myCar2= new Car(); myCar2.move(20);  
        myCar1.printPosition(); myCar2.printPosition();  
        myCar1.catchUp(myCar2);  
        myCar1.printPosition(); myCar2.printPosition();  
    }  
}
```

Παράδειγμα

- Φτιάξετε μια κλάση που να χειρίζεται ένα λογαριασμό τράπεζας. Κρατάει το όνομα του ιδιοκτήτη και το ποσό.
- Δημιουργείστε και μία μέθοδο που συγχωνεύει δύο λογαριασμούς του ίδιου ατόμου.

```
class BankAccount
{
    private String name;
    private int amount;

    public BankAccount(String name, int amount){
        this.name = name;
        this.amount = amount;
    }

    public void merge(BankAccount other){
        if (this.name.equals(other.name)) {
            this.amount += other.amount;
        }
    }
}
```

Είναι σύνηθες το αποτέλεσμα μιας μεθόδου να αποθηκεύει το αποτέλεσμα της στο ίδιο αντικείμενο το οποίο κάλεσε την μέθοδο.

Π.χ. εδώ το αποτέλεσμα της συγχώνευσης αποθηκεύεται στον λογαριασμό που έκανε την κλήση.

Αντικείμενα σαν ορίσματα – Παράδειγμα I

- Θέλουμε να προσομοιώσουμε την κυκλοφορία σε ένα δρόμο.
 - Έχουμε ένα φανάρι που μπορεί να είναι πράσινο, ή κόκκινο. Αλλάζει σε κάθε βήμα
 - Έχουμε ένα όχημα που σε κάθε βήμα κινείται μία θέση, αν το φανάρι δεν είναι κόκκινο.
- Κλάσεις:
 - **TrafficLight**: κρατάει την κατάσταση του φαναριού και αλλάζει την κατάσταση του
 - **Car**: Τροποποίηση της **move** ώστε παίρνει **όρισμα το φανάρι** και να κινείται μόνο αν το φανάρι δεν είναι κόκκινο.
 - **TrafficSimulation**: κάνει την προσομοίωση.

```
class TrafficLight
{
    boolean isLightRed = false;

    public void change(){
        isLightRed = !isLightRed;
    }

    public boolean isRed(){
        return isLightRed;
    }

    public void printStatus(){
        if (isLightRed){
            System.out.println(
                "Traffic light is red");
        }else{
            System.out.println(
                "Traffic light is green");
        }
    }
}
```

Το όρισμα στην περίπτωση αυτή είναι από άλλη κλάση, άρα δεν μπορούμε να δούμε τα πεδία του, πρέπει να καλέσουμε τη μέθοδο isRed()

```
class Car
{
    private int position = 0;

    public int printPosition() {
        System.out.println("Car at "+ position);
    }

    public void move(TrafficLight light){
        if (!light.isRed()){
            position ++;
        }
    }
}
```

```
class TrafficSimulation
{
    public static void main(String[] args){
        TrafficLight light = new TrafficLight();
        Car myCar = new Car();
        for (int i = 0; i < 10; i ++){
            light.printStatus();
            myCar.printPosition();
            myCar.move(light);
            light.change();
        }
    }
}
```


ΑΝΤΙΚΕΙΜΕΝΑ ΜΕ ΠΙΝΑΚΕΣ

Θα υλοποιήσετε την κλάση **DynamicArray**. Η κλάση θα κρατάει ένα πίνακα θετικών ακεραίων με τα στοιχεία του πίνακα, την χωρητικότητα (**capacity**) του πίνακα, και τον αριθμό των στοιχείων. Ο constructor παίρνει σαν όρισμα την αρχική χωρητικότητα και αρχικοποιεί τον πίνακα. Η κλάση θα πρέπει να έχει επίσης τις μεθόδους:

- **add**: Προσθέτει ένα νέο ακέραιο στο τέλος του πίνακα (υποθέστε ότι ο αριθμός είναι θετικός, δεν χρειάζεται να κάνετε τον έλεγχο). Αν ο πίνακας είναι γεμάτος, τότε αντικαθιστά τον πίνακα με έναν διπλάσιας χωρητικότητας, αντιγράφει τα στοιχεία και μετά κάνει την προσθήκη του νέου στοιχείου
- **remove**: Αφαιρεί το τελευταίο στοιχείο στον πίνακα και το επιστρέφει (αν δεν υπάρχουν στοιχεία, επιστρέφει -1). Αν μετά την αφαίρεση ο αριθμός των στοιχείων είναι το ένα τέταρτο της χωρητικότητας του πίνακα, τότε αντικαθιστά τον πίνακα με έναν της μισής χωρητικότητας και αντιγράφει τα στοιχεία.
- Την μέθοδο **πρόσβασης** (**accessor**) για τη χωρητικότητα του πίνακα.
- **print**: Τυπώνει τα στοιχεία του πίνακα.

Πεδία

- Τι πεδία χρειάζεται να κρατήσουμε?
 - Την χωρητικότητα `capacity` του πίνακα
 - Τον πίνακα από ακεραίους
 - Τον αριθμό από ακεραίους που έχουν αποθηκευτεί (`size`)
- Τι πρέπει να κάνει ο `constructor`?
 - Να δώσει τιμή στο `capacity`
 - Να δώσει χώρο στον πίνακα (`new`)

```
class DynamicArray
{
    public DynamicArray(int capacity)
    {
        int[] values = new int[capacity];
        int size = 0;
    }

    public void print()
    {
        for (int i = 0; i < size; i ++){
            System.out.println(values[i] + " ");
        }
        System.out.println();
    }
}
```

Σωστό ή λάθος?

Οι μεταβλητές capacity, size και values δεν είναι ορισμένες.

Για να μπορεί να τις βλέπει η μέθοδος print (ή οποιαδήποτε άλλη μέθοδος) θα πρέπει να είναι ορισμένες ως πεδία της κλάσης

ΛΑΘΟΣ!

```
class DynamicArray
{
    private int capacity;
    private int[] values;
    private int size = 0;

    public DynamicArray(int capacity)
    {
        int[] values = new int[capacity];
    }

    public void print()
    {
        for (int i = 0; i < size; i ++){
            System.out.println(values[i] + " ");
        }
        System.out.println();
    }
}
```

Σωστό?

Ο constructor δεν αρχικοποιεί τα πεδία της κλάσης .

Οι μεταβλητές **capacity** και **values** που ορίζονται μέσα στον constructor είναι **τοπικές μεταβλητές** και δεν αλλάζουν την τιμή των πεδίων.

ΛΑΘΟΣ!

```
class DynamicArray
{
    private int capacity;
    private int[] values;
    private int size = 0;

    public DynamicArray(int capacity)
    {
        this.capacity = capacity;
    }

    public void print()
    {
        for (int i = 0; i < size; i ++){
            System.out.println(values[i] + " ");
        }
        System.out.println();
    }
}
```

Σωστό?

Το capacity
αρχικοποιείται σωστά.

Ο πίνακας values όμως
όχι.

Τον έχουμε ορίσει σωστά
αλλά δεν του έχουμε
δώσει χώρο! Δεν έχουμε
προσδιορίσει το μέγεθος
του

ΛΑΘΟΣ!

```
class DynamicArray
{
    private int capacity;
    private int[] values = new int[capacity];
    private int size = 0;

    public DynamicArray(int capacity)
    {
        this.capacity = capacity;
    }

    public void print()
    {
        for (int i = 0; i < size; i++){
            System.out.println(values[i] + " ");
        }
        System.out.println();
    }
}
```

Σωστό?

Θυμηθείτε ότι οι εντολές αυτές θα εκτελεστούν πριν από τις εντολές του constructor. Εκείνη τη στιγμή δεν ξέρουμε το capacity είναι μηδέν και άρα δημιουργούμε ένα πίνακα μηδενικού μεγέθους!

ΛΑΘΟΣ!

```
class DynamicArray
{
    private int capacity;
    private int values[];
    private int size = 0;

    public DynamicArray(int capacity)
    {
        values = new int[capacity];
    }

    public void add(int x)
    {
        if (size == capacity){
            .....
        }
    }
}
```

Σωστό?

Ο Constructor θα αρχικοποιήσει σωστά τον πίνακα values, αλλά δεν θα αλλάξει το πεδίο capacity μιας και χρησιμοποιεί την τοπική μεταβλητή

Το capacity εδώ αναφέρεται στο πεδίο και έχει τιμή μηδέν, άρα ο έλεγχος θα βγαίνει αληθής

ΛΑΘΟΣ!


```
class DynamicArray
```

```
{
```

```
private int capacity;
```

```
private int values[];
```

```
private int size = 0;
```

```
public DynamicArray(int capacity)
```

```
{
```

```
    this.capacity = capacity;
```

```
    values = new int[capacity];
```

```
}
```

```
}
```

Σωστό?

Πρώτα δηλώνουμε τα πεδία μέσα στην κλάση

Μετά δίνουμε τιμή στη διάσταση και αφού πλέον ξέρουμε τη διάσταση δίνουμε χώρο στον πίνακα που θα κρατάει τις τιμές.

Τώρα μπορούμε και να κάνουμε και την αρχικοποίηση

ΣΩΣΤΟ!

Η μέθοδος add

- Η μέθοδος add προσθέτει ένα ακέραιο στον πίνακα και αν γεμίσει ο πίνακας διπλασιάζει το μέγεθος του πίνακα
- Τι σημαίνει αν γεμίσει ο πίνακας?
 - Το size (αριθμός στοιχείων) είναι ίδιο με το capacity (χωρητικότητα/μέγεθος) του πίνακα
- Τι σημαίνει διπλασιάζει το μέγεθος του πίνακα?
 - Οι πίνακες έχουν σταθερό μέγεθος
 - Πρέπει να δημιουργήσουμε ένα καινούριο πίνακα με διπλάσια χωρητικότητα
 - Να αντιγράψουμε τα στοιχεία που έχουμε σε αυτό τον πίνακα
 - Να κάνουμε το πεδίο values της κλάσης να δείχνει στον νέο πίνακα.
 - Να ενημερώσουμε το πεδίο capacity

Η μέθοδος add

```
public void add(int x)
{
    if (size == capacity) {
        capacity = 2*capacity;
        int[] temp = new int[capacity];
        for (int i = 0; i < size; i ++){
            temp[i] = values[i];
        }
        values = temp;
    }
    values[size] = x;
    size ++;
}
```

Ενημερώνει το πεδίο capacity

Αντιγράφει τα στοιχεία

Ενημερώνει το πεδίο values

Ενημερώνει το πεδίο size

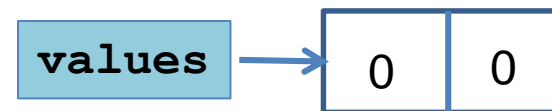
Προσθέτει το νέο στοιχείο στον πίνακα. Το πεδίο size μας δίνει και τον αριθμό των στοιχείων στον πίνακα αλλά και την επόμενη αδειανή θέση.

Δημιουργία νέου πίνακα

- Τι ακριβώς γίνεται όταν δημιουργούμε το νέο πίνακα?
- Η εντολή στον constructor

```
values = new int[capacity];
```

δημιουργεί χώρο στη μνήμη και βάζει το πεδίο `values` να δείχνει σε αυτόν.



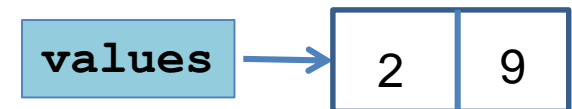
Δημιουργία νέου πίνακα

- Τι ακριβώς γίνεται όταν δημιουργούμε το νέο πίνακα?
- Η εντολή στον constructor

```
values = new int[capacity];
```

δημιουργεί χώρο στη μνήμη και βάζει το πεδίο `values` να δείχνει σε αυτόν.

- Μετά από δύο `add`

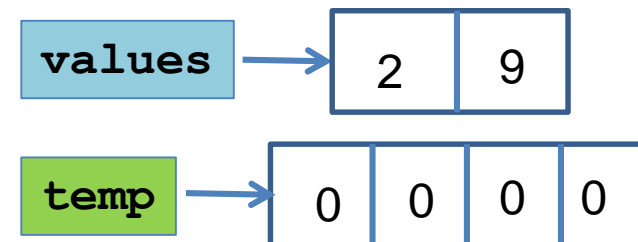


Δημιουργία νέου πίνακα

- Η εντολή στην add

```
int[] temp = new int[2*capacity];
```

δημιουργεί νέο χώρο στη μνήμη και βάζει την τοπική μεταβλητή `temp` να δείχνει σε αυτόν.



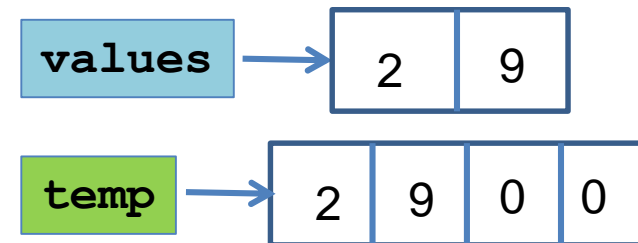
Δημιουργία νέου πίνακα

- Η εντολή στην add

```
int[] temp = new int[2*capacity];
```

δημιουργεί νέο χώρο στη μνήμη και βάζει την τοπική μεταβλητή `temp` να δείχνει σε αυτόν.

- Με το for loop αντιγράφουμε τα στοιχεία από το `values` στο `temp`



Δημιουργία νέου πίνακα

- Η εντολή στην add

```
int[] temp = new int[2*capacity];
```

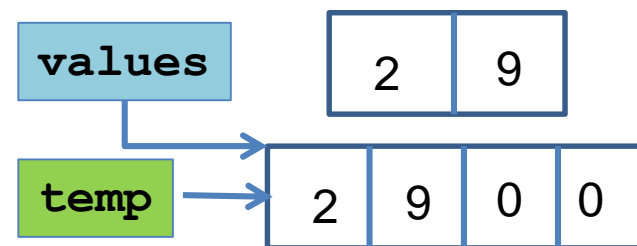
δημιουργεί νέο χώρο στη μνήμη και βάζει την τοπική μεταβλητή `temp` να δείχνει σε αυτόν.

- Με το for loop αντιγράφουμε τα στοιχεία από το `values` στο `temp`

- Με την εντολή

```
values = temp
```

Βάζουμε το πεδίο `values` να δείχνει στον χώρο που δείχνει η μεταβλητή `temp`



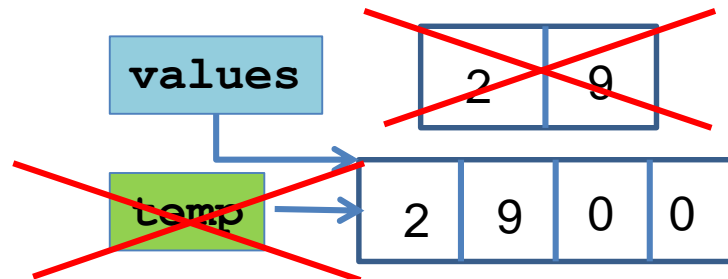
Δημιουργία νέου πίνακα

- Με την εντολή

`values = temp`

Βάζουμε το πεδίο `values` να δείχνει στον χώρο που δείχνει η μεταβλητή `temp`

- Ο προηγούμενος χώρος χάνεται.
- Το `values` είναι πεδίο και άρα ο νέος χώρος διατηρείται και μετά το τέλος της `add` αφού εξαφανιστεί η `temp`.



Η μέθοδος remove

- Η remove αφαιρεί το τελευταίο στοιχείο και το επιστρέφει. Αν ο αριθμός των στοιχείων πέσει στο $\frac{1}{4}$ υποδιπλασιάζει το μέγεθος του πίνακα
- Τι σημαίνει αφαιρεί?
 - Ο αριθμός των στοιχείων μας λέει πόσα στοιχεία «βλέπουμε». Αν μειώσουμε τον αριθμό κατά 1, ουσιαστικά αφαιρούμε το τελευταίο στοιχείο. Δεν χρειάζεται να το «σβήσουμε».

Η μέθοδος remove

```
public int remove()
{
    if (size == 0) {
        return -1;
    }
    size -- ;
    int retValue = values[size];
    if (size == capacity/4) {
        capacity = capacity/2;
        int[] temp = new int[capacity];
        for (int i = 0; i < size; i ++){
            temp[i] = array[i];
        }
        values = temp;
    }
    return retValue;
}
```

Ελέγχει την περίπτωση του άδειου πίνακα

Ενημερώνει το πεδίο size

Κρατάει το στοιχείο που θέλουμε να επιστρέψουμε. Είναι στην θέση size-1

Ενημερώνει το πεδίο capacity

Δημιουργεί πίνακα μισού μεγέθους και αντιγράφει τα στοιχεία

Ενημερώνει το πεδίο values

Επιστρέφει την τιμή

Accessor για το capacity

- Οι accessor μέθοδοι (όπως και οι mutator) έχουν ένα πολύ **συγκεκριμένο όνομα**, και **συγκεκριμένη λειτουργία**
- Όνομα:
 - `get<Όνομα Πεδιου>()`
 - Στην περίπτωση μας `getCapacity()`
- Λειτουργία:
 - **Επιστρέφουν** την τιμή του πεδίου. **Δεν εκτυπώνουν!**

```
public int getCapacity()  
{  
    return capacity;  
}
```

Η κλάση DynamicArrayTest

```
class DynamicArrayTest
{
    public static void main(String[] args) {
        //DynamicArray array = new DynamicArray(2);
        //System.out.println("Removed:"+array.remove());
        //array.add(2);
        //array.add(4);
        //array.add(9);
        // Εκτυπώστε το capacity του array
        //array.print();
        //System.out.println("Removed:"+array.remove());
        //System.out.println("Removed:"+array.remove());
        // Εκτυπώστε το capacity του array
        //array.print();
    }
}
```

Τα προγράμματα σας πρέπει να τα δημιουργείτε κομμάτι-κομμάτι.
Κάθε φορά που ολοκληρώνετε μια μέθοδο πρέπει να την τεστάρετε.
Αυτός ήταν ο στόχος της DynamicArrayTest

ΟΙ ΜΕΘΟΔΟΙ TOSTRING ΚΑΙ EQUALS

Δυο ειδικές μέθοδοι

- Η Java «περιμένει» να δει τις εξής δύο μεθόδους για κάθε αντικείμενο
 - Τη μέθοδος `toString` η οποία για ένα αντικείμενο επιστρέφει μία `string` αναπαράσταση του αντικειμένου.
 - Τη μέθοδο `equals` η οποία ελέγχει για ισότητα δύο αντικειμένων
- Και οι δύο συναρτήσεις ορίζονται από τον προγραμματιστή
 - Το τι `String` θα επιστραφεί και τι σημαίνει δύο αντικείμενα να είναι ίσα μπορούν να οριστούν όπως μας βολεύει.

toString και equals

- Η μέθοδος toString ορίζεται **πάντα** ως:

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

- Αν έχουμε ορίσει την toString μπορούμε να χρησιμοποιήσουμε **τα αντικείμενα της κλάσης σαν Strings**
 - Καλείται αυτόματα η toString

- Η μέθοδος equals ορίζεται **πάντα** ως:

```
public boolean equals (<Class name> other) {  
    ...  
}
```


Παράδειγμα

- Στην κλάση `Car` θέλουμε να προσθέσουμε τις μεθόδους `toString` και `equals`
 - Η `toString` θα επιστρέφει ένα `String` με τη θέση του αυτοκινήτου
 - Η `equals` θα ελέγχει αν δύο οχήματα έχουν την ίδια θέση.

toString()

```
class Car
{
    private Integer position = 0;

    public Car(int position){
        this.position = position;
    }

    public void move(int delta){
        position += delta ;
    }

    public String toString(){
        return position.toString();
    }
}

class MovingCarToString
{
    public static void main(String args[]){
        Car myCar1 = new Car(1);
        Car myCar2 = new Car(0); myCar2.move(2);
        System.out.println("Car 1 is at " + myCar1 + " and car 2 is at " + myCar2);
    }
}
```

Για να μπορούμε να μετατρέψουμε τον ακέραιο σε String ορίζουμε το position ως **Integer** (wrapper class)

Η Java περιμένει αυτό το συντακτικό για τον ορισμό της **toString**

Μετά καλούμε τη συνάρτηση **toString()** της κλάσης **Integer**

Χρησιμοποιούμε τις myCar1, myCar2 σαν String. Καλείται η μέθοδος toString() αυτόματα

Ισοδύναμο με το:

```
System.out.println("Car 1 is at " + myCar1.toString() + " and car 2 is at " + myCar2.toString());
```

toString()

```
class Car
{
    private int position = 0;

    public Car(int position){
        this.position = position;
    }

    public void move(int delta){
        position += delta ;
    }

    public String toString(){
        return ""+position;
    }
}

class MovingCarToString
{
    public static void main(String args[]){
        Car myCar1 = new Car(1);
        Car myCar2 = new Car(0); myCar2.move(2);
        System.out.println("Car 1 is at " + myCar1 + " and car 2 is at " + myCar2);
    }
}
```

Ένας άλλος τρόπος να μετατρέψουμε ένα int σε String

```
class Car
{
    private int position = 0;

    public Car(int position){
        this.position = position;
    }

    public void move(int delta){
        position += delta ;
    }

    public boolean equals(Car other){
        if (this.position == other.position) {
            return true;
        }
        return false;
    }
}
```

Η Java περιμένει αυτό το συντακτικό
για τον ορισμό της **equals**

Ένα παράδειγμα αντικειμένου
ως παράμετρος συνάρτησης

Αν και το πεδίο position είναι private
μπορούμε να το προσπελάσουμε γιατί
είμαστε μέσα στην κλάση Car.
Μία κλάση μπορεί να προσπελάσει τα
ιδιωτικά μέλη όλων των αντικειμένων
της κλάσης

Χρήση της **return** για
έλεγχο ροής

```
class MovingCarEquals
{
    public static void main(String args[]){
        Car myCar1 = new Car(2);
        Car myCar2 = new Car(0); myCar2.move(2);
        if (myCar1.equals(myCar2)) {
            System.out.println("Collision!");
        }
    }
}
```

Κλήση της **equals** στο
πρόγραμμα

Παράδειγμα

- Πως θα ορίσουμε τις μεθόδους `toString` και `equals` για την κλάση `Person`?

```
class Person
{
    private String name;

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

    public String toString(){
        return name;
    }

    public boolean equals(Person other){
        return this.name.equals(other.name);
    }
}

public class TwoPersons
{
    public static void main(String[] args){
        Person alice = new Person("Alice");
        Person bob = new Person("Bob");
        if (!alice.equals(bob)){
            System.out.println("There are two different persons: "
                + alice + "and " + bob);
        }
    }
}
```

```
class Person{
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String toString(){
        return firstName + " " + lastName;
    }

    public boolean equals(Person other){
        return (this.firstName.equals(other.firstName))
            && (this.lastName.equals(other.lastName));
    }
}

public class TwoPersons2
{
    public static void main(String[] args){
        Person alice = new Person("Alice", "Wonderland");
        Person bob = new Person("Bob", "Sfougkarakis");
        if (!alice.equals(bob)){
            System.out.println("There are two different persons: "
                + alice + "and " + bob);
        }
    }
}
```

ΑΝΤΙΚΕΙΜΕΝΑ ΜΕΣΑ ΣΕ ΑΝΤΙΚΕΙΜΕΝΑ

Αντικείμενα μέσα σε αντικείμενα

- Εκτός από ορίσματα σε μεθόδους αντικείμενα οποιαδήποτε κλάσης μπορούν να εμφανιστούν και ως πεδία μιας κλάσης
 - Ένα αντικείμενο μπορεί να έχει μέσα του άλλα αντικείμενα.

```
class Person
```

```
{  
    private String name;  
  
    public Person(String name){  
        this.name = name;  
    }  
  
    public String getName(){  
        return name;  
    }  
}
```

```
class Car
```

```
{  
    private int position = 0;  
    private Person driver;  
  
    public Car(int position, String name){  
        this.position = position;  
        this.driver = new Person(name);  
    }  
  
    public String toString(){  
        return driver.getName()  
            + " " + position;  
    }  
}
```

```
class MovingCarDriver1
```

```
{  
    public static void main(String args[])  
    {  
        Car myCar = new Car(1, "Alice");  
        System.out.println(myCar);  
    }  
}
```

Το αντικείμενο δημιουργείται μέσα στον constructor. Αυτό έχει νόημα αν το Person χρησιμοποιείται μόνο μέσα στην κλάση Car.

```
class Person
```

```
{  
    private String name;  
  
    public Person(String name){  
        this.name = name;  
    }  
  
    public String getName(){  
        return name;  
    }  
}
```

```
class Car
```

```
{  
    private int position = 0;  
    private Person driver;  
  
    public Car(int position, Person driver){  
        this.position = position;  
        this.driver = driver;  
    }  
  
    public String toString(){  
        return driver.getName()  
            + " " + position;  
    }  
}
```

```
class MovingCarDriver2
```

```
{  
    public static void main(String args[])  
    {  
        Person alice = new Person("Alice");  
        Car myCar = new Car(1, alice);  
        System.out.println(myCar);  
    }  
}
```

Καλύτερη υλοποίηση!

```
class Person
```

```
{  
    private String name;  
    private int age;  
  
    public Person(String name,  
                  int age){  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public int getAge(){  
        return age;  
    }  
}
```

Η Person είναι διαφορετική κλάση
άρα δεν μπορούμε να διαβάσουμε
το πεδίο age

```
class Car
```

```
{  
    private int position = 0;  
    private Person driver;  
  
    public Car(int position, Person driver){  
        this.position = position;  
        if (driver.getAge() >= 18){  
            this.driver = driver;  
        }  
    }  
  
    public String toString(){  
        return driver.getName()  
            + " " + position;  
    }  
}
```

```
class MovingCarDriver
```

```
{  
    public static void main(String args[])  
    {  
        Person alice = new Person("Alice");  
        Car myCar = new Car(1, alice);  
        System.out.println(myCar);  
    }  
}
```

Η εντολή `exit`

```
class Car
{
    private int position = 0;
    private Person driver;

    public Car(int position, Person driver) {
        this.position = position;
        if (driver.getAge() >= 18) {
            this.driver = driver;
        }
        else{
            System.exit(-1);
        }
    }
}
```

Χρησιμοποιείται για σοβαρά λάθη για να σταματάει την εκτέλεση του προγράμματος.

Αν δώσουμε μη αποδεκτή ηλικία το πρόγραμμα μας θα σταματήσει.

Το -1 εξυπηρετεί σαν κωδικός λάθους, μπορείτε να βάλετε όποια τιμή θέλετε.

```
class Person
```

```
{  
    private String name;  
    private int licence;  
  
    public Person(String name,  
                   int licence){  
        this.name = name;  
        this.licence = licence;  
    }  
}
```

```
class Car
```

```
{  
    private int position = 0;  
    private Person driver;  
  
    public Car(int position, Person driver){  
        this.position = position;  
        this.driver = driver;  
    }  
}
```

Πως θα υλοποιήσουμε την `toString` και την `equals`?

```
class Person
{
    private String name;
    private int licence;

    public Person(String name,
                  int licence){
        this.name = name;
        this.licence = licence;
    }

    public String toString(){
        return name + " " + licence;
    }

    public boolean equals(Person other){
        if (this.name.equals(other.name)&&
            this.licence == other.licence)){
            return true
        }else{
            return false;
        }
    }
}
```

```
class Car
{
    private int position = 0;
    private Person driver;

    public Car(int position, Person driver){
        this.position = position;
        this.driver = driver;
    }

    public String toString(){
        return driver + " " + position;
    }

    public boolean equals(Car other){
        if (this.position == other.position &&
            this.driver.equals(other.driver)){
            return true;
        }else{
            return false;
        }
    }
}
```

Φωλιασμένη κλήση της toString
και της equals

Φωλιασμένες toString και equals

- Ένα αντικείμενο μπορεί να περιέχει μέσα άλλα αντικείμενα.
- Είναι συνηθισμένο σε αυτή την περίπτωση η `toString` και `equals` να ορίζονται κάνοντας **φωλιασμένη** κλήση της `toString` και της `equals` των αντικειμένων που περιέχει.
- **Φωλιασμένη** κλήση της `toString` της κλάσης `driver`

```
public String toString(){  
    return driver + " " + position;  
}
```

- **Φωλιασμένη** κλήση της `toString` της κλάσης `driver`

```
public boolean equals(Car other){  
    if (this.position == other.position &&  
        this.driver.equals(other.driver)) {  
        return true;  
    }else{  
        return false;  
    }  
}
```


Κώδικας σε πολλά αρχεία

- Όταν έχουμε πολλές κλάσεις βολεύει να τις βάζουμε σε **διαφορετικά αρχεία**.
 - Το κάθε αρχείο έχει το όνομα της κλάσης
 - Σημείωση: μια κλάση μόνη της σε ένα αρχείο είναι by default public, μαζί με άλλη είναι by default private.
- Ένα επιπλέον πλεονέκτημα είναι ότι μπορούμε να ορίσουμε μια **main** συνάρτηση για κάθε κλάση ξεχωριστά
 - Βοηθάει για το testing του κώδικα.
- Για να κάνουμε compile πολλά αρχεία μαζί:
 - **javac file1.java file2.java file3.java**
 - ή μπορούμε να κάνουμε compile το “**βασικό**” αρχείο

ΑΝΤΙΚΕΙΜΕΝΑ ΩΣ ΕΠΙΣΤΡΕΦΟΜΕΝΕΣ ΤΙΜΕΣ

Αντικείμενα ως επιστρεφόμενες τιμές

- Μία μέθοδος μπορεί να επιστρέφει αντικείμενα όπως οποιαδήποτε άλλη τιμή.
- Είναι δυνατόν επίσης μέσα σε μία μέθοδο να δημιουργούμε ένα αντικείμενο και να το επιστρέφουμε για να χρησιμοποιηθεί μετά.

```
class Car
{
    private int position = 0;
    private Person driver;

    public Car(int position, String name) {
        this.position = position;
        this.driver = new Person(name);
    }

    public String toString() {
        return driver.getName()
            + " " + position;
    }

    public Person getDriver() {
        return driver;
    }
}
```

Επιστρέφει το αντικείμενο Person το οποίο είναι ο οδηγός του οχήματος.

```
class BankAccount
```

```
{  
    private String name;  
    private int amount;
```

Μια άλλη επιλογή είναι να δημιουργήσουμε ένα νέο λογαριασμό μετά την συγχώνευση

```
    public BankAccount(String name, int amount) {  
        this.name = name;  
        this.amount = amount;  
    }
```

```
    public void merge(BankAccount other) {  
        if (this.name.equals(other.name)) {  
            this.amount += other.amount;  
        }  
    }
```

Δημιουργούμε ένα νέο αντικείμενο BankAccount και το επιστρέφουμε.

```
    public BankAccount mergeIntoNewAccount(BankAccount other) {  
        if (this.name.equals(other.name)) {  
            BankAccount newAccount =  
                new BankAccount(name, this.amount+other.amount);  
            return newAccount;  
        }  
        return null;  
    }  
}
```

Αν δεν μπορούμε να δημιουργήσουμε το νέο λογαριασμό επιστρέφουμε **null**. Το null είναι το κενό αντικείμενο.

Παράδειγμα

- Κάνετε μια μέθοδο για την κλάση `Car` η οποία να παίρνει σαν όρισμα ένα αριθμό `K` και να κάνει `K` κινήσεις με τυχαίο αριθμό από βήματα στο διάστημα μεταξύ 0 έως 9

```
import java.util.Random;

class Car
{
    private int position = 0;

    public int[] move(int K){
        int[] moves = new int[K];
        Random rnd = new Random();
        for (int i = 0; i < K; i ++){
            int delta = rnd.nextInt(10);
            position += delta ;
            moves[i] = delta;
        }
        return moves;
    }

    public void printPosition(){
        System.out.println("Car is at position "+position);
    }
}

class MovingCar14
{
    public static void main(String args[]){
        Car myCar = new Car();
        int[] moves = myCar.move(4);
        myCar.printPosition();
        System.out.print("moves: ");
        for (int i = 0; i < 4; i ++){
            System.out.print(moves[i]+" ");
        }
        System.out.println();
    }
}
```

ΑΝΤΙΚΕΙΜΕΝΑ ΜΕ ΠΙΝΑΚΕΣ II

Παράδειγμα

- Δημιουργήστε μια κλάση `CarWithPassengers` η οποία μπορεί να κρατάει στο όχημα πολλαπλούς επιβάτες, αντικείμενα `Person`.

Η CAR_SIZE είναι σταθερά για την κλάση

Δημιουργία πίνακα από αντικείμενα Person

```
class CarWithPassengers
```

```
{
```

```
private int CAR_SIZE = 4;
```

```
private int position = 0;
```

```
private Person[] passengers = new Person[CAR_SIZE];
```

```
private int numOfPassengers = 0;
```

```
public void addPassenger(Person passenger)
```

```
{
```

```
    if (numOfPassengers < CAR_SIZE) {
```

```
        passengers[numOfPassengers] = passenger;
```

```
        numOfPassengers ++;
```

```
    }
```

```
}
```

toString με φωλιασμένες κλήσεις

```
public String toString(){
```

```
    String retValue = "Car at "+position +" with passengers: ";
```

```
    for (int i = 0; i < numOfPassengers; i ++){
```

```
        retValue += passengers[i]+ " ";
```

```
    }
```

```
    return retValue;
```

```
}
```

```
}
```

Μπορούμε να τεστάρουμε το πρόγραμμα με μία main στην CarWithPassengers

```
class Person
```

```
{
```

```
    private String name;
```

```
    public Person(String name){
```

```
        this.name = name;
```

```
    }
```

```
    public String toString(){
```

```
        return name;
```

```
    }
```

```
}
```

```
public static void main(String[] args){
```

```
    CarWithPassengers myCar = new CarWithPassengers();
```

```
    Person alice = new Person("Alice");
```

```
    myCar.addPassenger(alice);
```

```
    Person bob = new Person("Bob");
```

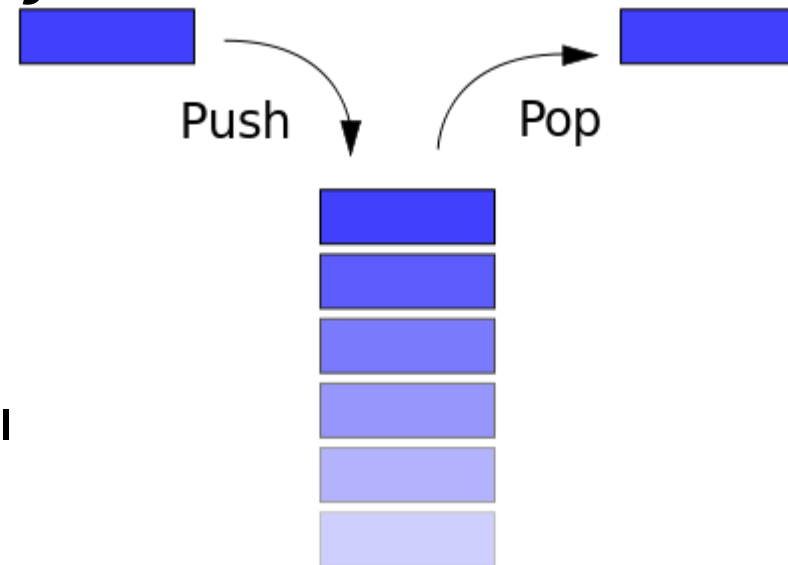
```
    myCar.addPassenger(bob);
```

```
    System.out.println(myCar);
```

```
}
```

Παράδειγμα ADT: Στοίβα (Stack)

- Η **Στοίβα** είναι μια συλλογή δεδομένων η οποία επιτρέπει τις εξής λειτουργίες:
 - **push(element)**: προσθέτει ένα νέο στοιχείο στην **κορυφή της στοίβας**
 - **pop()**: αφαιρεί και επιστρέφει το στοιχείο το οποίο βρίσκεται στην **κορυφή της στοίβας**.
 - **isEmpty()**: **ελέγχει** αν η στοίβα είναι **άδεια** και επιστρέφει true ή false
- Η Στοίβα υλοποιεί την πολιτική **Last-In-First-Out (LIFO)** στη σειρά που μας δίνει τα στοιχεία
 - Χρήσιμο σε διάφορες εφαρμογές, π.χ., για τη δέσμευση μνήμης στην κλήση συναρτήσεων



Υλοποίηση

- Θα υλοποιήσουμε μια Στοίβα ακεραίων χρησιμοποιώντας ένα **πίνακα** (Στοιβα συγκεκριμένης χωρητικότητας)
 - Τι πεδία πρέπει να ορίσουμε?
 - Τι μεθόδους?

```
class Stack
{
    private int capacity;
    private int size = 0;
    private int[] elements;

    public Stack(int capacity){
        this.capacity = capacity;
        elements = new int[capacity];
    }

    public void push(int element){
        if (size == capacity){
            System.out.println("Cannot enter any more elements");
            return;
        }
        elements[size] = element;
        size ++;
    }

    public int pop(){
        if (size == 0){
            System.out.println("No elements to pop");
            return -1;
        }
        size -- ;
        return elements[size];
    }

    public boolean isEmpty(){
        return (size == 0);
    }
}
```

Εφαρμογές

- Υπολόγισε την δυαδική μορφή ενός ακεραίου.

```
class Binary
{
    public static void main(String[] args)
    {
        Stack myStack = new Stack(100);
        int number = 1973;

        while (number > 0) {
            myStack.push(number%2);
            number = number/2;
        }

        while (!myStack.isEmpty()) {
            System.out.print(myStack.pop());
        }
    }
}
```

ΕΠΕΚΤΑΣΕΙΣ

- Πως θα ορίσουμε την μέθοδο equals?
- Πως θα ορίσουμε τη μέθοδο toString?


```
public String toString(){
    String returnString = "";
    for (int i = 0; i < size; i ++){
        returnString = returnString + elements[i] + " ";
    }
    return returnString;
}

public boolean equals(Stack other){
    if (this.size != other.size){
        return false;
    }
    for (int i = 0; i < size; i ++){
        if (this.elements[i] != other.elements[i]){
            return false;
        }
    }
    return true;
}
```

6. ΑΝΑΦΟΡΕΣ

Αναφορές

Στοίβα και Σωρός

Αναφορές-Παράμετροι

String Interning

Μέθοδοι που επιστρέφουν αντικείμενα

Βαθιά και ρηχά αντίγραφα

Copy Constructor

Η αναφορά this

Σύνθεση και αναφορές

new

- Όπως είδαμε για να δημιουργήσουμε ένα αντικείμενο χρειάζεται να καλέσουμε τη **new**.
 - Για τον πίνακα είπαμε ότι έτσι δίνουμε χώρο στον πίνακα και δεσμεύουμε την απαιτούμενη μνήμη.
 - Για τα αντικείμενα δεσμεύουμε τον απαραίτητο χώρο για το αντικείμενο.
- Τι ακριβώς συμβαίνει όταν καλούμε την new?

Η μνήμη του υπολογιστή

- Η **κύρια μνήμη** (main memory) του υπολογιστή κρατάει τα **δεδομένα** (και τις εντολές) για την εκτέλεση των προγραμμάτων.
 - Η μνήμη είναι προσωρινή, τα δεδομένα χάνονται όταν ολοκληρωθεί το πρόγραμμα.
- Η μνήμη είναι χωρισμένη σε **bytes** (8 bits)
 - Ο χώρος που χρειάζεται για ένα **χαρακτήρα ASCII**.
- Το κάθε byte έχει μια **διεύθυνση**, με την οποία μπορούμε να προσπελάσουμε τη συγκεκριμένη θέση μνήμης
 - **Random Access Memory (RAM)**
 - Σε 32-bit συστήματα μια διεύθυνση είναι 32 bits, σε 64-bit συστήματα μια διεύθυνση είναι 64 bits.

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	'a'
0001	'b'
0010	'c'
0011	'd'
0100	'e'
0101	'f'
0110	'g'
0111	'h'

Αποθήκευση μεταβλητών

- Η **κύρια μνήμη** (main memory) του υπολογιστή κρατάει τις **μεταβλητές** ενός προγράμματος
- Μια μεταβλητή μπορεί να απαιτεί χώρο περισσότερο από 1 byte.
 - Π.χ., οι μεταβλητές τύπου double χρειάζονται 8 bytes.
 - Η μεταβλητή τότε αποθηκεύεται σε συνεχόμενα bytes στη μνήμη.
- Η **θέση μνήμης** (διεύθυνση) της μεταβλητής θεωρείται το **πρώτο byte** από το οποίο ξεκινάει η αποθήκευση του της μεταβλητής.
 - Στο παράδειγμα μας η μεταβλητή βρίσκεται στη θέση 0000
 - Αν ξέρουμε την αρχή και το μέγεθος της μεταβλητής μπορούμε να τη διαβάσουμε.
- Άρα μία **θέση μνήμης** αποτελείται από μία **διεύθυνση** και το **μέγεθος**.

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	8.5
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Αποθήκευση μεταβλητών πρωταρχικού τύπου

- Για τις μεταβλητές **πρωταρχικού** τύπου (char, int, double,...) ξέρουμε εκ των προτέρων το μέγεθος της μνήμης που χρειαζόμαστε.
- Όταν ο μεταγλωττιστής δει τη **δήλωση** μιας μεταβλητής πρωταρχικού τύπου **δεσμεύει** μια θέση μνήμης αντίστοιχου μεγέθους
 - Η δήλωση μιας μεταβλητής ουσιαστικά **δίνει ένα όνομα** σε μία θέση μνήμης
 - Συχνά λέμε η **θέση μνήμης x** για τη μεταβλητή **x**.

```
int x = 5;  
int y = 3;
```

	Διεύθυνση μνήμης	Περιεχόμενο μνήμης
x	0000	5
	0001	
	0010	
	0011	
y	0100	3
	0101	
	0110	
	0111	

Αποθήκευση αντικειμένων

- Για τα αντικείμενα δεν ξέρουμε πάντα εκ των προτέρων το μέγεθος της μνήμης που θα πρέπει να δεσμεύσουμε.

```
String s; // δεν ξερούμε το μέγεθος του s  
s = "ab"; // το s έχει μέγεθος 2 χαρακτήρες  
s = "abc"; // το s έχει μέγεθος 3 χαρακτήρες
```

- Παρομοίως αν δηλώσουμε

```
int[] A;
```

μας λέει ότι έχουμε ένα πίνακα από ακέραιους αλλά δεν μας λέει πόσο μεγάλος θα είναι αυτός ο πίνακας.

```
A = new int[2];  
A = new int[3];
```

Αποθήκευση αντικειμένων

- Οι θέσεις μνήμης των αντικειμένων κρατάνε μια **διεύθυνση** στο χώρο στον οποίο αποθηκεύεται το αντικείμενο
- Η διεύθυνση αυτή λέγεται **αναφορά**.
- Οι αναφορές είναι παρόμοιες με τους **δείκτες** σε άλλες γλώσσες προγραμματισμού με τη διαφορά ότι η Java δεν μας αφήνει να πειράξουμε τις διευθύνσεις.
 - Εμείς χρησιμοποιούμε μόνο τη μεταβλητή του αντικειμένου, όχι το την διεύθυνση που περιέχει
- Το **dereferencing** το κάνει η Java αυτόματα.

```
String s = "ab";
```

	Διεύθυνση μνήμης	Περιεχόμενο μνήμης
s	0000	0100
	0001	
	0010	
	0011	
	0100	a
	0101	b
	0110	
	0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Η δεσμευμένη λέξη **null** σημαίνει μια **κενή αναφορά** (μια διεύθυνση που δεν δείχνει πουθενά)

	Διεύθυνση μνήμης	Περιεχόμενο μνήμης
A	0000	null
	0001	
	0010	
	0011	
	0100	
	0101	
	0110	
	0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Με την εντολή **new** δεσμεύουμε δύο θέσεις ακεραίων και η αναφορά του A δείχνει σε αυτό το χώρο που δεσμεύσαμε

A

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0011
0001	
0010	
0011	0
0100	0
0101	
0110	
0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Ο τελεστής [] για τον πίνακα μας πάει στην αντίστοιχη θέση του χώρου που κρατήσαμε

A

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0011
0001	
0010	
0011	10
0100	0
0101	
0110	
0111	

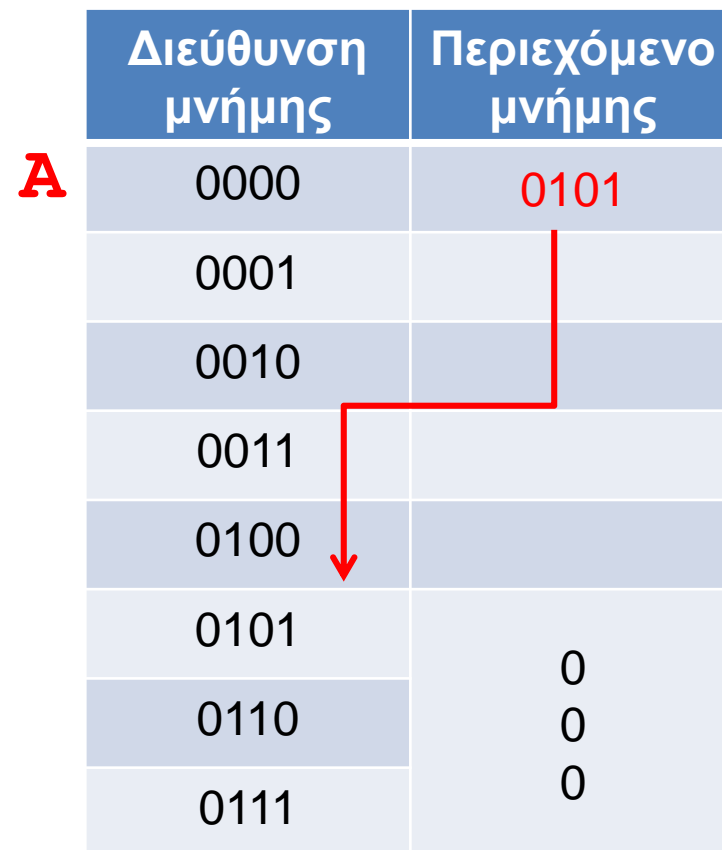
Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Με νέα κλήση της **new** δεσμεύουμε νέο χώρο για το A, και αν δεν έχουμε κρατήσει την προηγούμενη αναφορά σε κάποια άλλη μεταβλητή τότε χάνεται (garbage collection), όπως και οι τιμές που είχαμε αποθηκεύσει στον πίνακα.

A

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0101
0001	
0010	
0011	
0100	
0101	
0110	0
0111	0



Αντικείμενα κλάσεων

- Τι γίνεται με τα αντικείμενα κλάσεων που ορίσαμε εμείς?
- Παράδειγμα: Η κλάση `Person` (ToyClass από το βιβλίο).

```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber) {
        name = initName;
        number = initNumber;
    }

    public void set(String newName, int newNumber) {
        name = newName;
        number = newNumber;
    }

    public String toString() {
        return (name + " " + number);
    }
}
```

Παράδειγμα

```
Person varP = new Person ("Bob", 1);
```

varP

Η κλήση της **new** δημιουργεί ένα χώρο μνήμης για την αποθήκευση του αντικειμένου τύπου **Person** το οποίο κρατάει ένα **string** και ένα ακέραιο (δεσμεύεται χώρος και γι αυτά).

Η μεταβλητή **varP** κρατάει την διεύθυνση του χώρου στην μνήμη όπου αποθηκεύσαμε αυτό το αντικείμενο.

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	
0010	"Bob"
0011	
0100	
0101	
0110	
0111	

Αναθέσεις μεταξύ αντικειμένων

Τι θα τυπώσει το παρακάτω πρόγραμμα?

```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Αναθέσεις μεταξύ αντικειμένων

varP1

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	
0010	"Bob"
0011	
0100	1
0101	
0110	
0111	

```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

Αναθέσεις μεταξύ αντικειμένων

varP1

varP2

```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	null
0010	"Bob"
0011	
0100	
0101	1
0110	
0111	

Αναθέσεις μεταξύ αντικειμένων

Η ανάθεση του `varP1` στο `varP2` έχει αποτέλεσμα η μεταβλητή `varP2` να δείχνει στην ίδια θέση μνήμης όπως και η `varP1`

`varP1`

`varP2`

```
Person varP1 = new Person("Bob", 1);
Person varP2;
varP2 = varP1;
varP2.set("Ann", 2);
System.out.println(varP1);
```

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	0010
0010	"Bob"
0011	
0100	
0101	1
0110	
0111	

Αναθέσεις μεταξύ αντικειμένων

Η αλλαγή θα γίνει στο χώρο μνήμης που δείχνει ο `varP2`
Αυτός είναι ο ίδιος όπως αυτός που δείχνει και ο `varP1`

`varP1`

`varP2`

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	0010
0010	"Ann"
0011	
0100	
0101	
0110	
0111	

```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

Αναθέσεις μεταξύ αντικειμένων

Τυπώνει "Ann 2"

Αλλάζοντας τα περιεχόμενα της θέσης μνήμης στην οποία δείχνει ο `varP2` αλλάζουμε και το `varP1`

```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

`varP1`

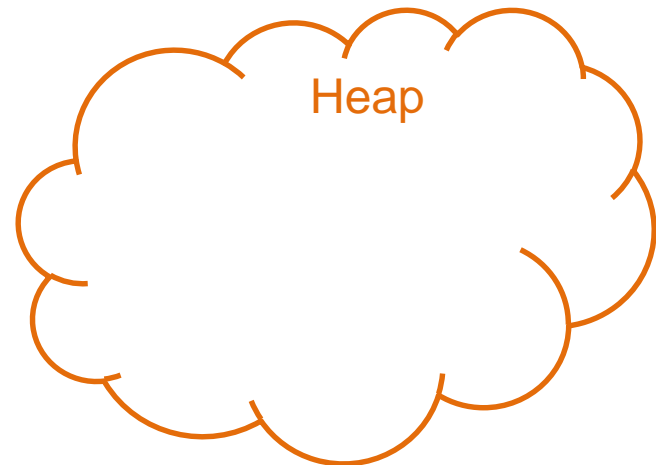
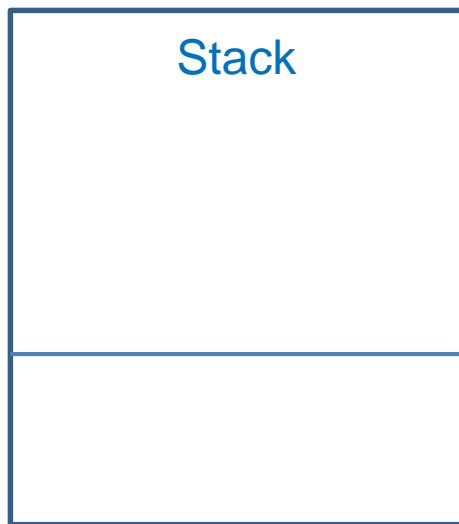
`varP2`

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	0010
0010	0010
0011	"Ann"
0100	
0101	2
0110	
0111	

ΣΤΟΙΒΑ ΚΑΙ ΣΩΡΟΣ

Διαχείριση μνήμης από το JVM

- Η μνήμη χωρίζεται σε δύο τμήματα
 - Τη στοίβα (**stack**) που χρησιμοποιείται για να κρατάει πληροφορία για τις **τοπικές μεταβλητές** κάθε μεθόδου/block.
 - Το σωρό (**heap**) που χρησιμοποιείται για να δεσμεύουμε **μνήμη για τα αντικείμενα**



Stack

- Κάθε φορά που καλείται μία μέθοδος, δημιουργείται ένα «πλαίσιο» (**frame**) για την μέθοδο στη στοίβα
 - Δημιουργείται ένας **χώρος μνήμης** που αποθηκεύει τις **παραμέτρους** και τις **τοπικές μεταβλητές** της μεθόδου.
- Αν η μέθοδος καλέσει μία άλλη μέθοδο θα δημιουργηθεί ένα νέο πλαίσιο και θα τοποθετηθεί (push) στην **κορυφή της στοίβας**.
- Όταν βγούμε από την μέθοδο το πλαίσιο **αφαιρείται** (pop) από την κορυφή της στοίβας και επιστρέφουμε στην προηγούμενη μέθοδο
- Στη βάση της στοίβας είναι η μέθοδος **main**.

Παράδειγμα

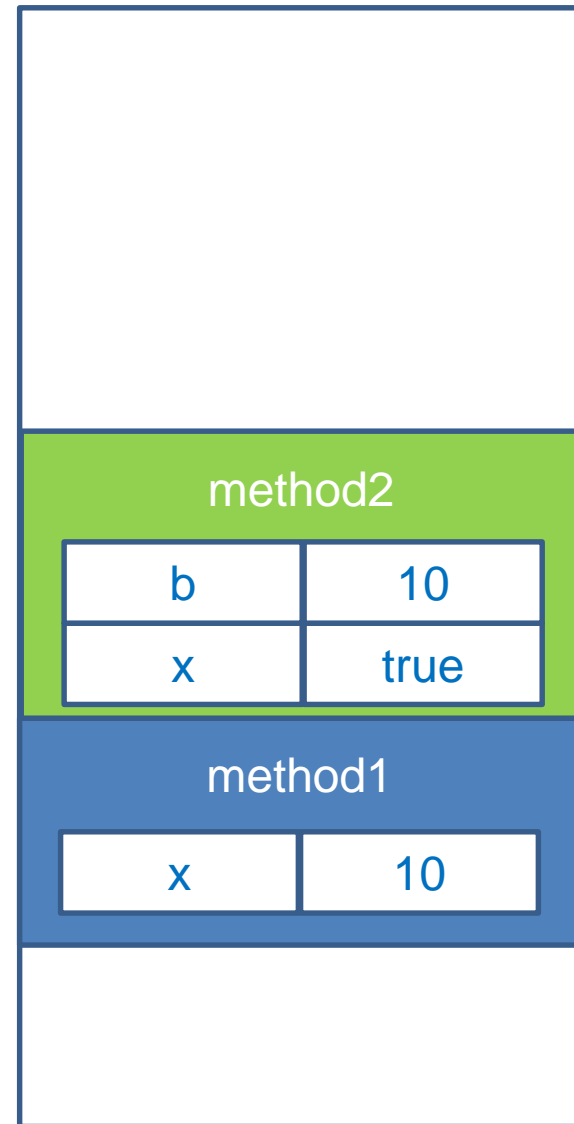
```
public void method1() {  
    int x = 10;  
    method2(x);  
}
```



Παράδειγμα

```
public void method1() {  
    int x = 10;  
    method2(x);  
}
```

```
public void method2(int b) {  
    boolean x = true;  
    method3();  
}
```

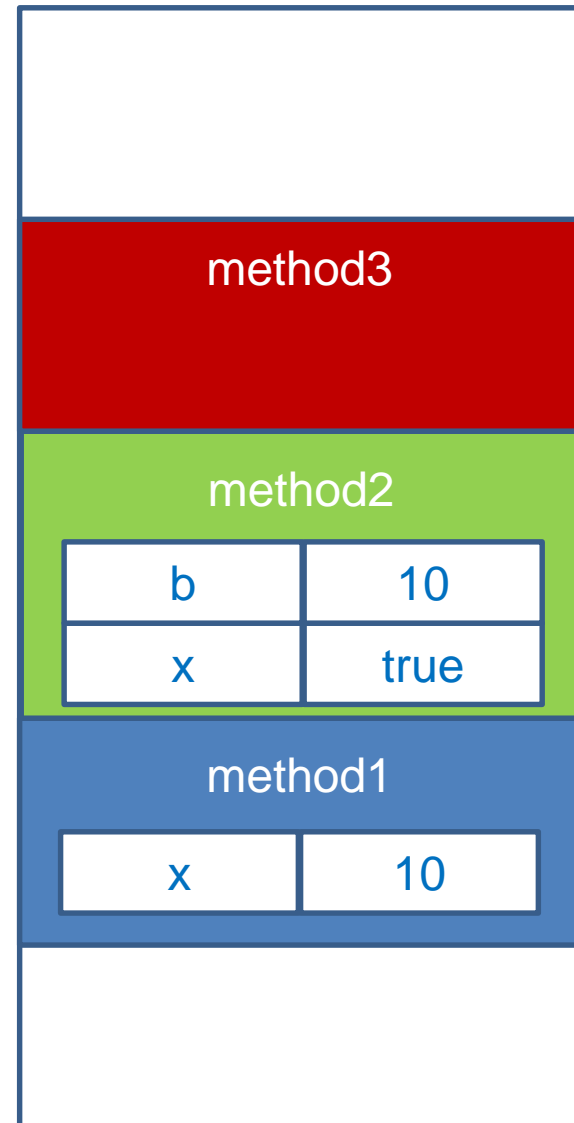


Παράδειγμα

```
public void method1 () {  
    int x = 10;  
    method2 (x) ;  
}
```

```
public void method2 (int b) {  
    boolean x = true;  
    method3 () ;  
}
```

```
public void method3 ()  
{...}
```



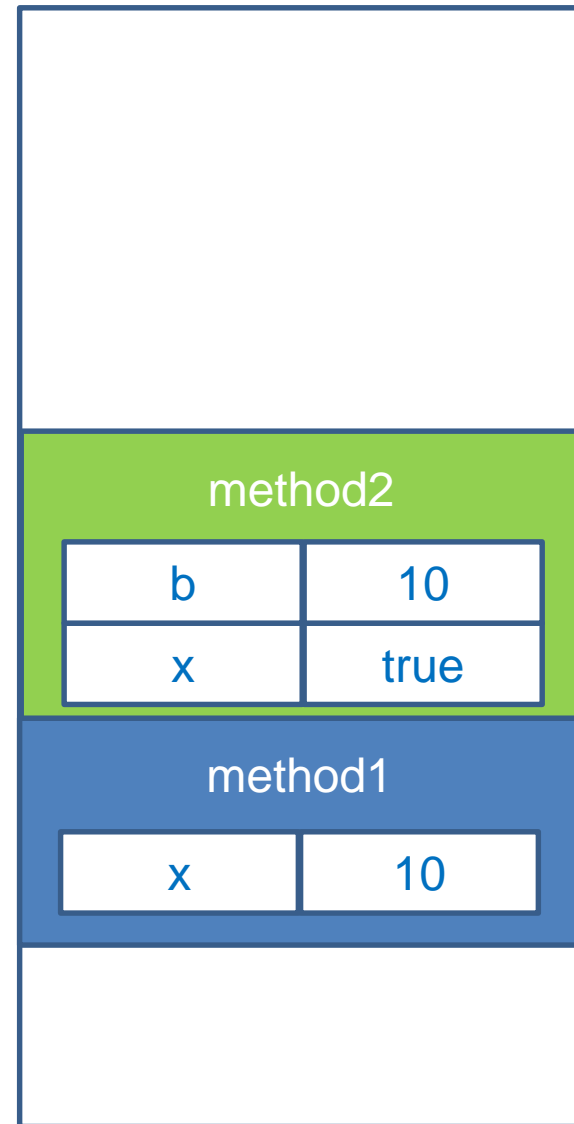
Παράδειγμα

```
public void method1() {  
    int x = 10;  
    method2(x);  
    method3();  
}
```



Παράδειγμα

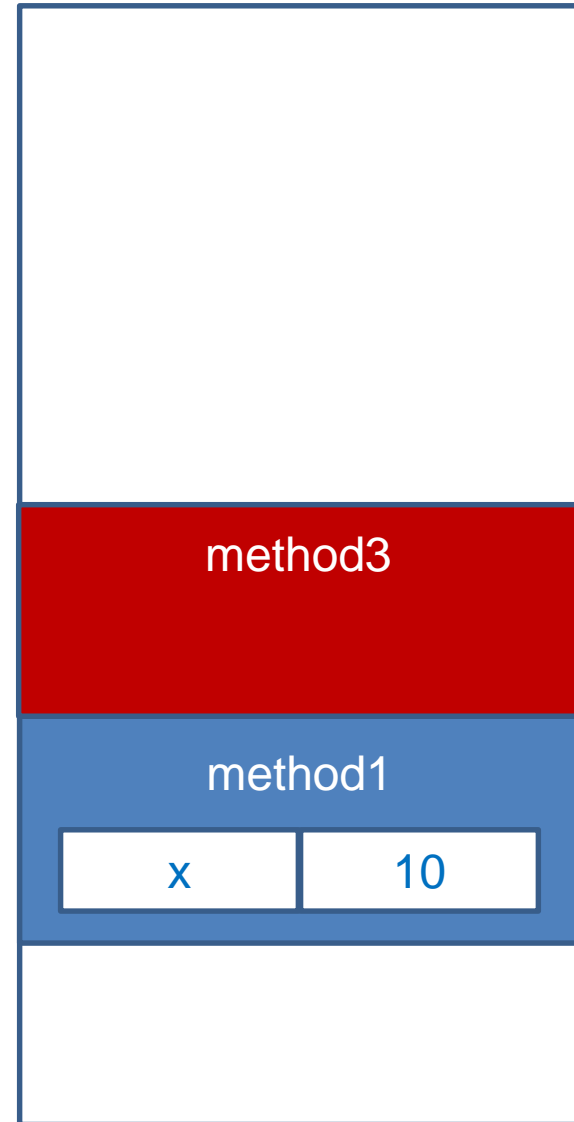
```
public void method1() {  
    int x = 10;  
    method2(x);  
    method3()  
}  
  
public void method2(int b) {  
    boolean x = (b==10);  
    ...  
}
```



Παράδειγμα

```
public void method1 () {  
    int x = 10;  
    method2 (x) ;  
}
```

```
public void method3 ()  
{...}
```



Heap

- Όταν μέσα σε μία μέθοδο δημιουργούμε ένα αντικείμενο με την **new** γίνονται τα εξής
 - στο πλαίσιο (frame) της μεθόδου (στη στοίβα) υπάρχει μια **τοπική μεταβλητή** που κρατάει την **αναφορά** στο αντικείμενο
 - Η κλήση της **new** δεσμεύει **χώρο μνήμης** στο σωρό (heap) για να κρατήσει τα πεδία του αντικειμένου.
 - Η **αναφορά** δείχνει στη **θέση μνήμης** που δεσμεύτηκε.

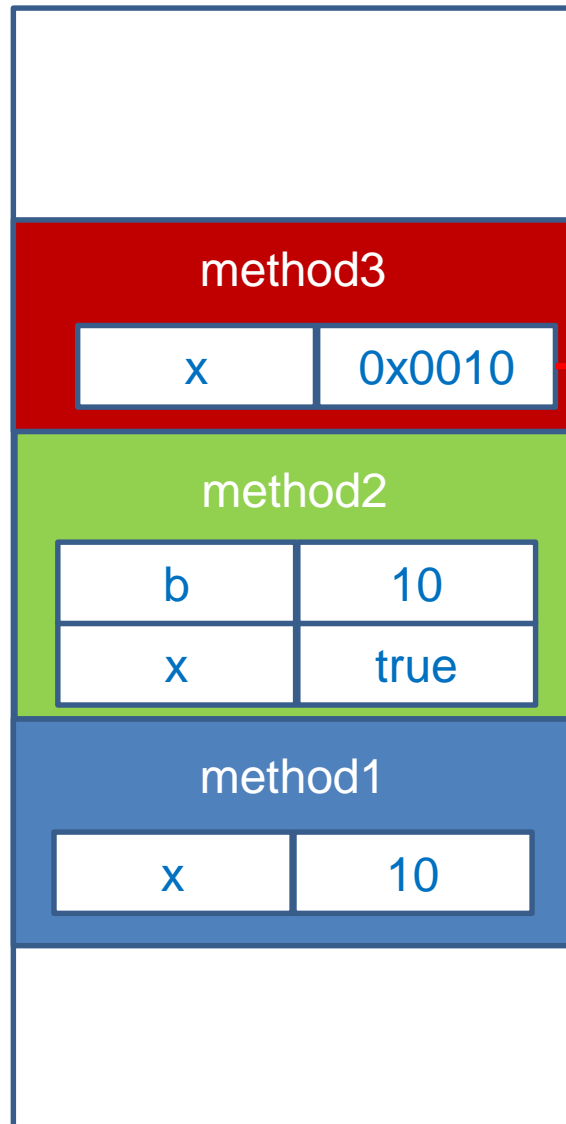

```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber) {
        name = initName;
        number = initNumber;
    }

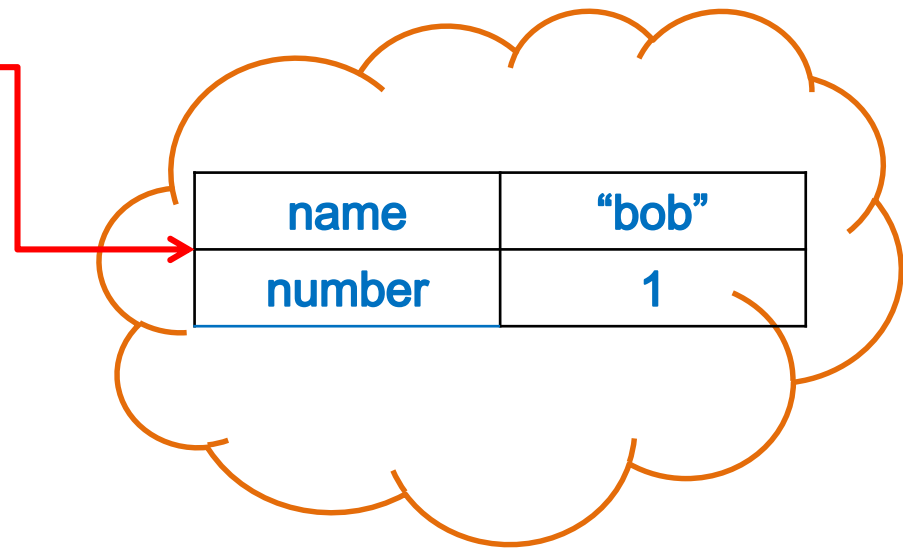
    public void set(String name, int number) {
        this.name = name;
        this.number = number;
    }

    public String toString() {
        return (name + " " + number);
    }
}
```

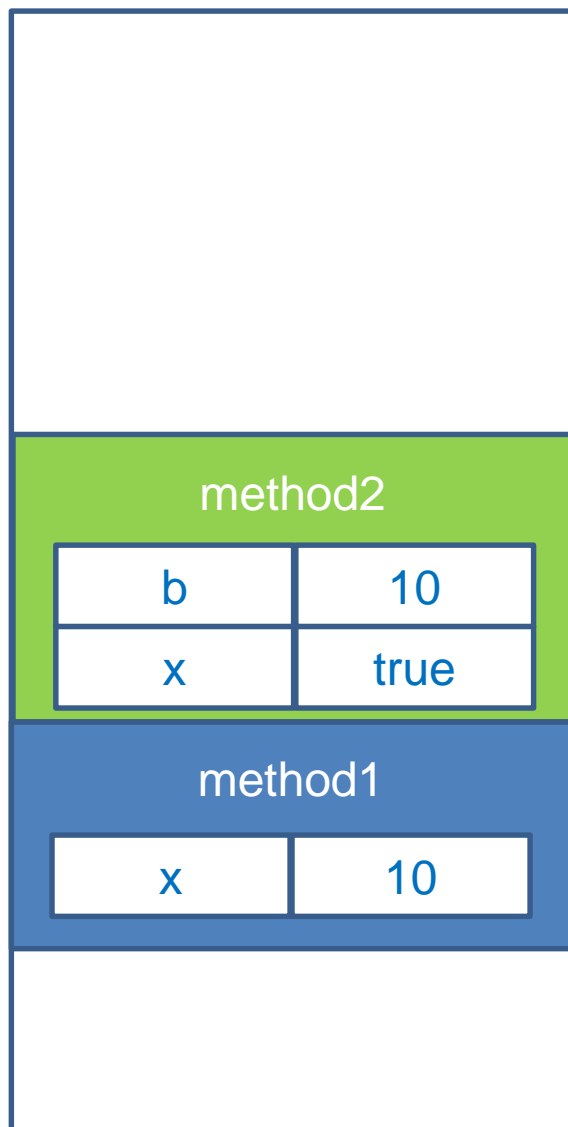
Παράδειγμα



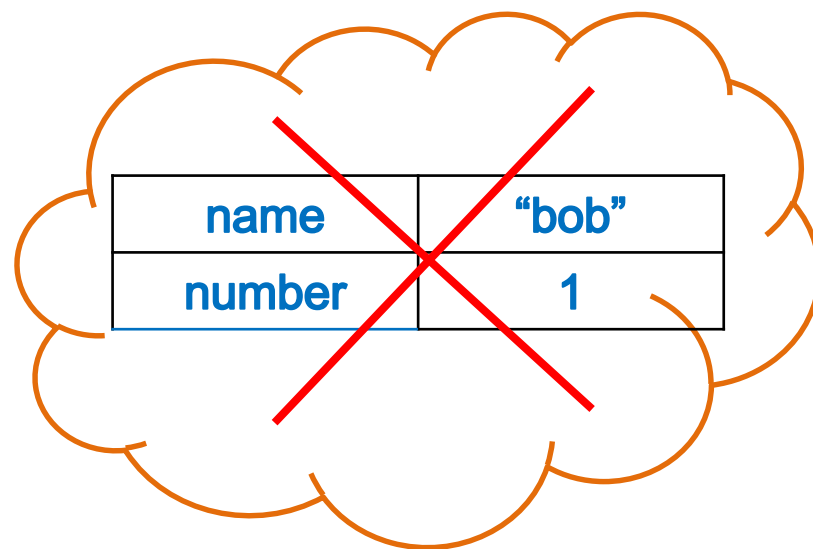
```
public void method3()  
{  
    Person x = new Person("bob",1);  
}
```



Παράδειγμα

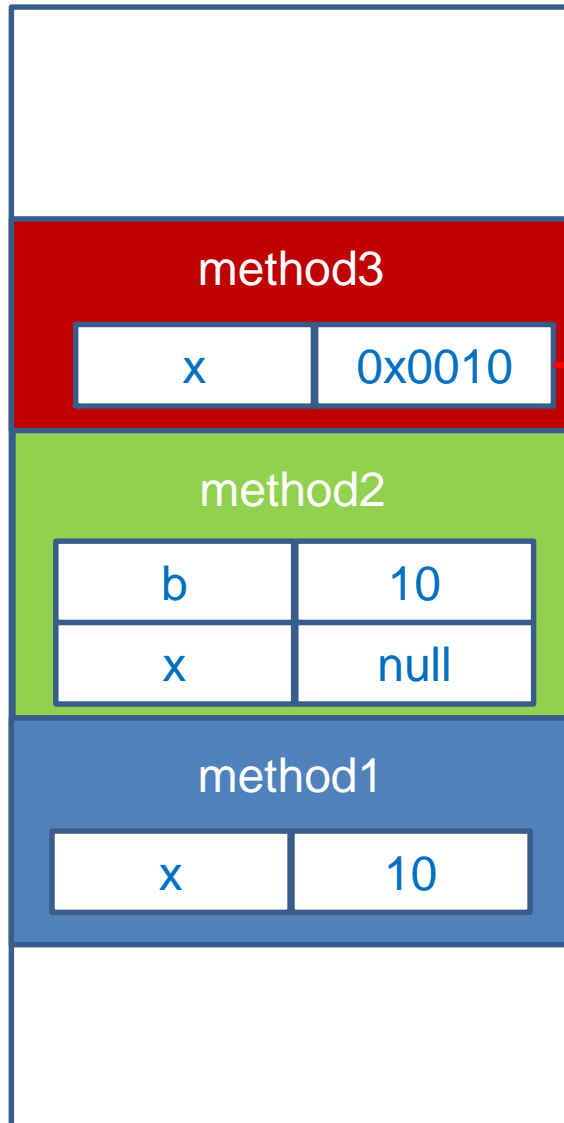


Όταν επιστρέφουμε από την μέθοδο method3 η αναφορά προς το αντικείμενο Person παύει να υπάρχει.

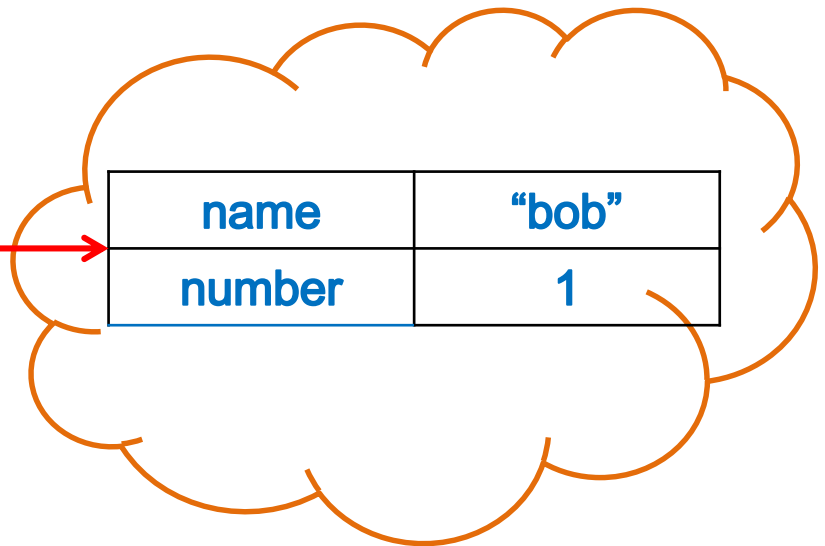


Αν δεν υπάρχουν άλλες αναφορές στο αντικείμενο τότε ο garbage collector αποδεσμεύει τη μνήμη του αντικειμένου

Παράδειγμα



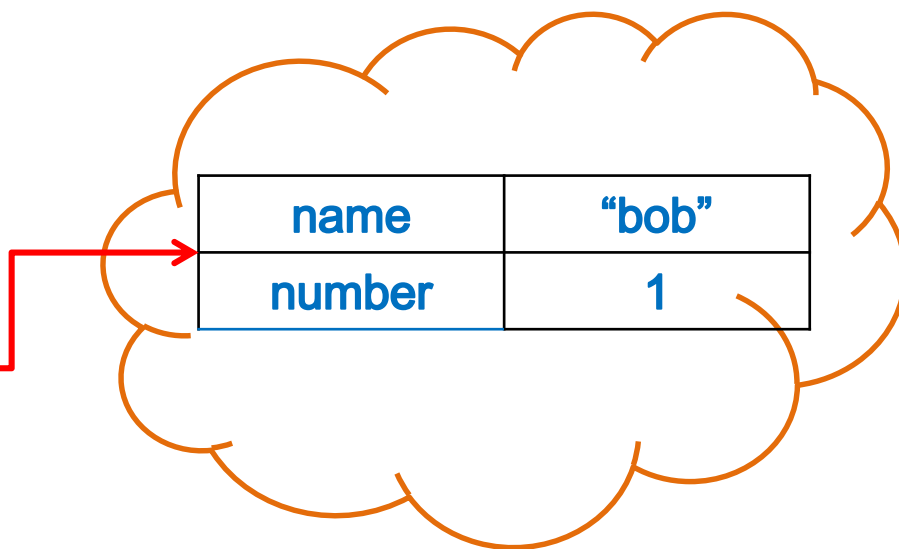
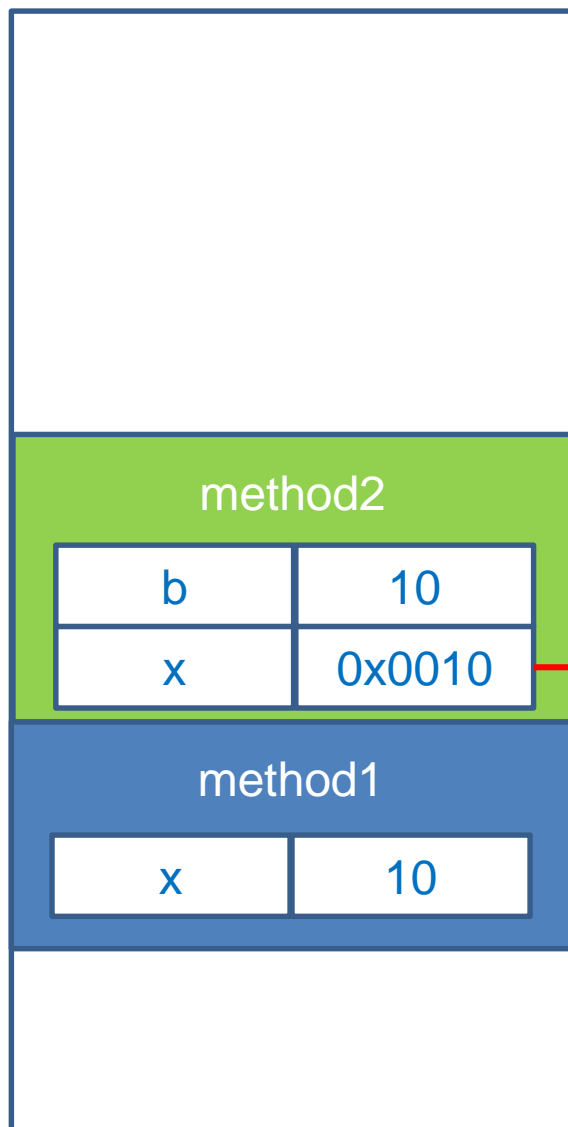
```
public Person method3()  
{  
    Person x = new Person("bob",1);  
    return x;  
}
```



Παράδειγμα

```
public void method2 ()  
{  
    Person x = method3 ()  
}
```

Στην περίπτωση αυτή η αναφορά στο αντικείμενο επιστρέφεται και αποθηκεύεται στην μεταβλητή x μέθοδο method2

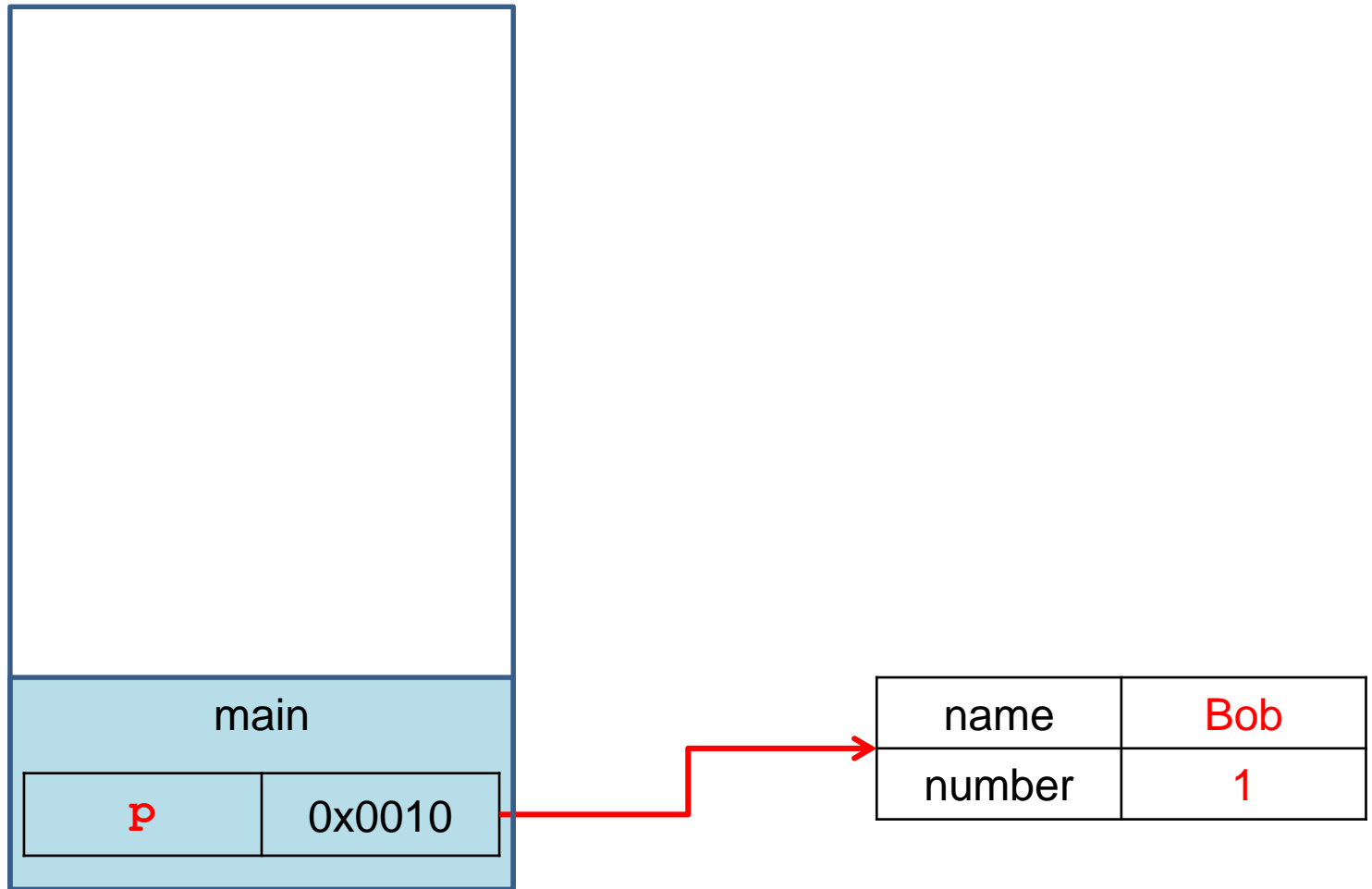


Η αναφορά δεν χάνεται και το αντικείμενο διατηρείται όσο υπάρχει αναφορά σε αυτό

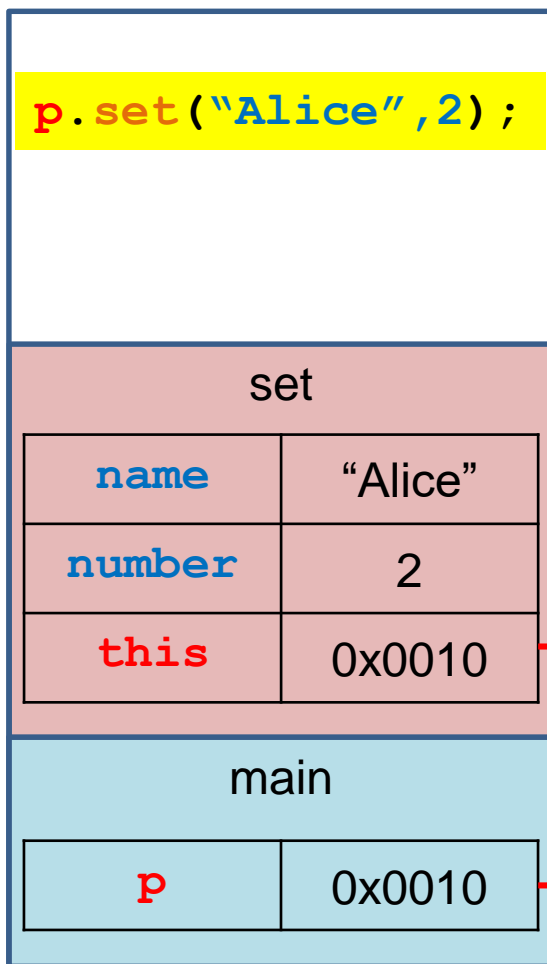
Κλήση μεθόδου από αντικείμενο

```
public class ObjectMethodCallDemo
{
    public static void main(String[] args)
    {
        Person p = new Person("Bob", 1);
        p.set("Alice", 2);
        System.out.println(p);
    }
}
```

Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος

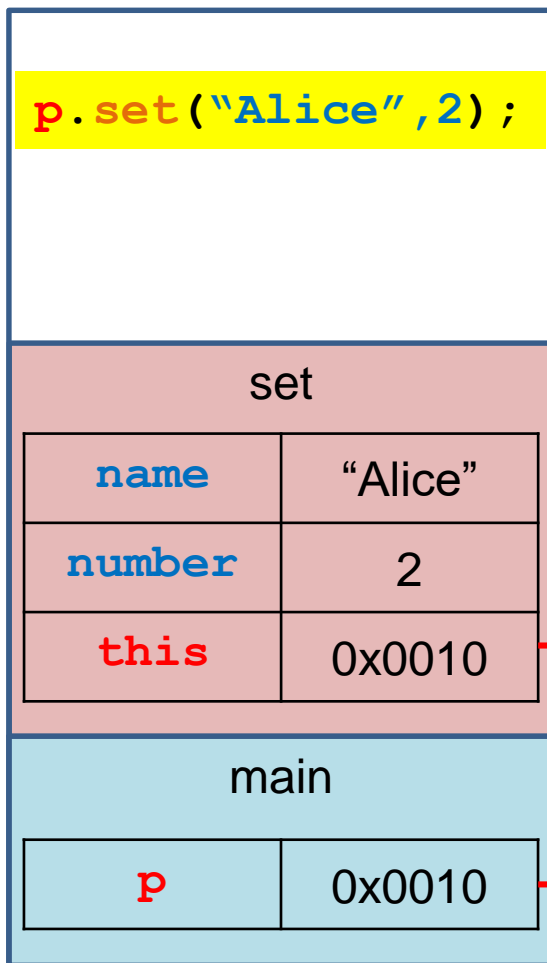


Όταν καλείται μια μέθοδος ενός αντικειμένου αυτόματα δημιουργείται στο frame της μεθόδου και η μεταβλητή **this** η οποία κρατάει μια αναφορά στο αρχικό αντικείμενο που κάλεσε την μέθοδο.

Την μεταβλητή αυτή μπορούμε να την χρησιμοποιήσουμε σαν οποιαδήποτε άλλη μεταβλητή.

name	Bob
number	1

Εξέλιξη του προγράμματος



```
this.name = name;  
this.number = number;
```

Τα this.name, this.number αναφέρονται στα πεδία του αντικειμένου ενώ τα name, number στις τοπικές μεταβλητές.

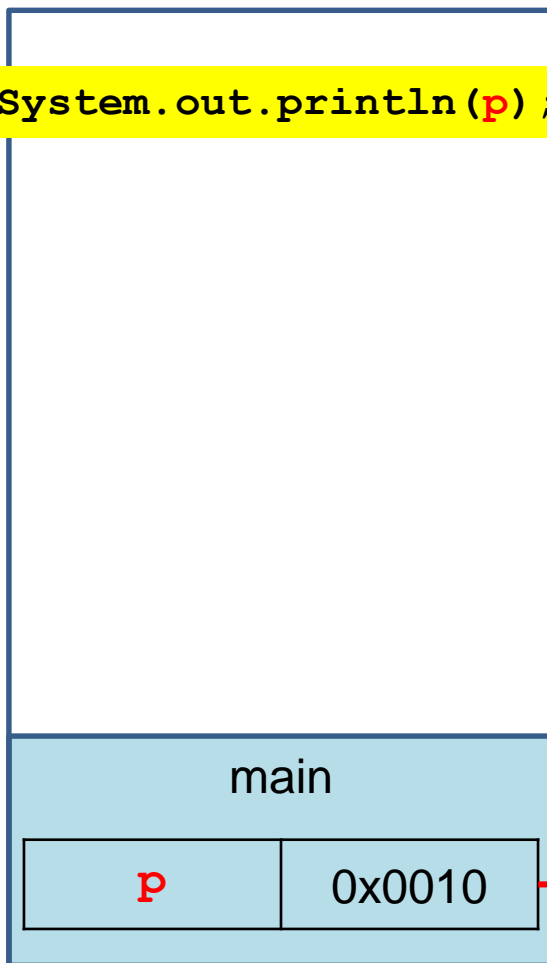
name	"Alice"
number	2

Εξέλιξη του προγράμματος

```
System.out.println(p);
```

Επιστρέφοντας οι αλλαγές που κάναμε στα πεδία του αντικειμένου this διατηρούνται στο χώρο μνήμης του p.

Τυπώνει "Alice 2"



name	"Alice"
number	2

ΑΝΤΙΚΕΙΜΕΝΑ ΩΣ ΠΑΡΑΜΕΤΡΟΙ

Αντικείμενα ως παράμετροι

- Όταν περνάμε παραμέτρους σε μία μέθοδο το πέρασμα γίνεται πάντα **δια τιμής (call-by-value)**
 - Δηλαδή απλά περνάμε τα **περιεχόμενα της θέσης μνήμης** της συγκεκριμένης μεταβλητής.
 - Για μεταβλητές **πρωταρχικού** τύπου, αλλαγές στην τιμή της παραμέτρου **δεν αλλάζουν** την μεταβλητή που περάσαμε σαν όρισμα.
- Τι γίνεται όμως αν η παράμετρος είναι ένα αντικείμενο?
 - Τα **περιεχόμενα της θέσης μνήμης** μιας μεταβλητής-αντικείμενο είναι μια **αναφορά**.
 - **Αν** μέσα στην μέθοδο **αλλάξουν τα περιεχόμενα του αντικειμένου** (εκεί που δείχνει η αναφορά) τότε **αλλάζει και η μεταβλητή-αντικείμενο** που περάσαμε.

```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber){
        name = initName;
        number = initNumber;
    }

    public void set(String newName, int newNumber){
        name = newName;
        number = newNumber;
    }

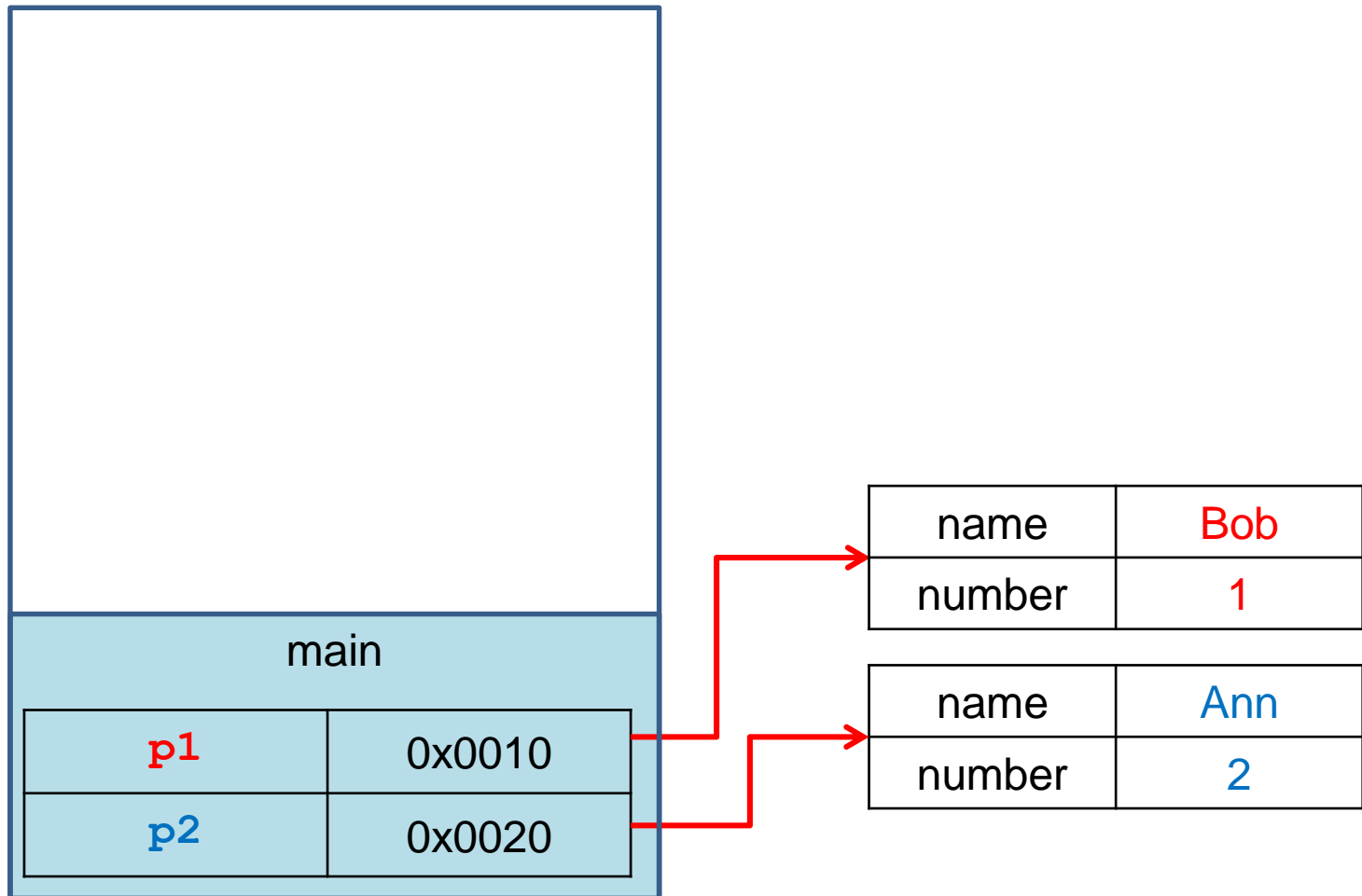
    public void copier(Person other) {
        other.name = this.name;
        other.number = this.number;
    }

    public String toString( ){
        return (name + " " + number);
    }
}
```

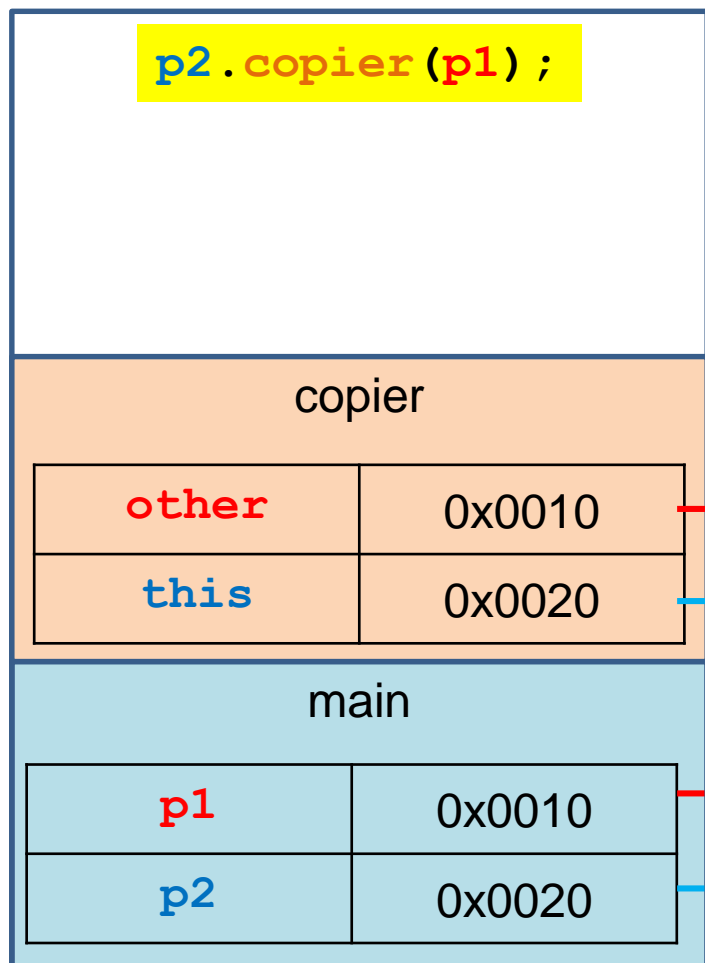
Παράδειγμα

```
public class ClassParameterDemo1
{
    public static void main(String[] args)
    {
        Person p1 = new Person("Bob", 1);
        Person p2 = new Person("Ann", 2);
        p2.copier(p1);
        System.out.println(p1);
    }
}
```

Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος



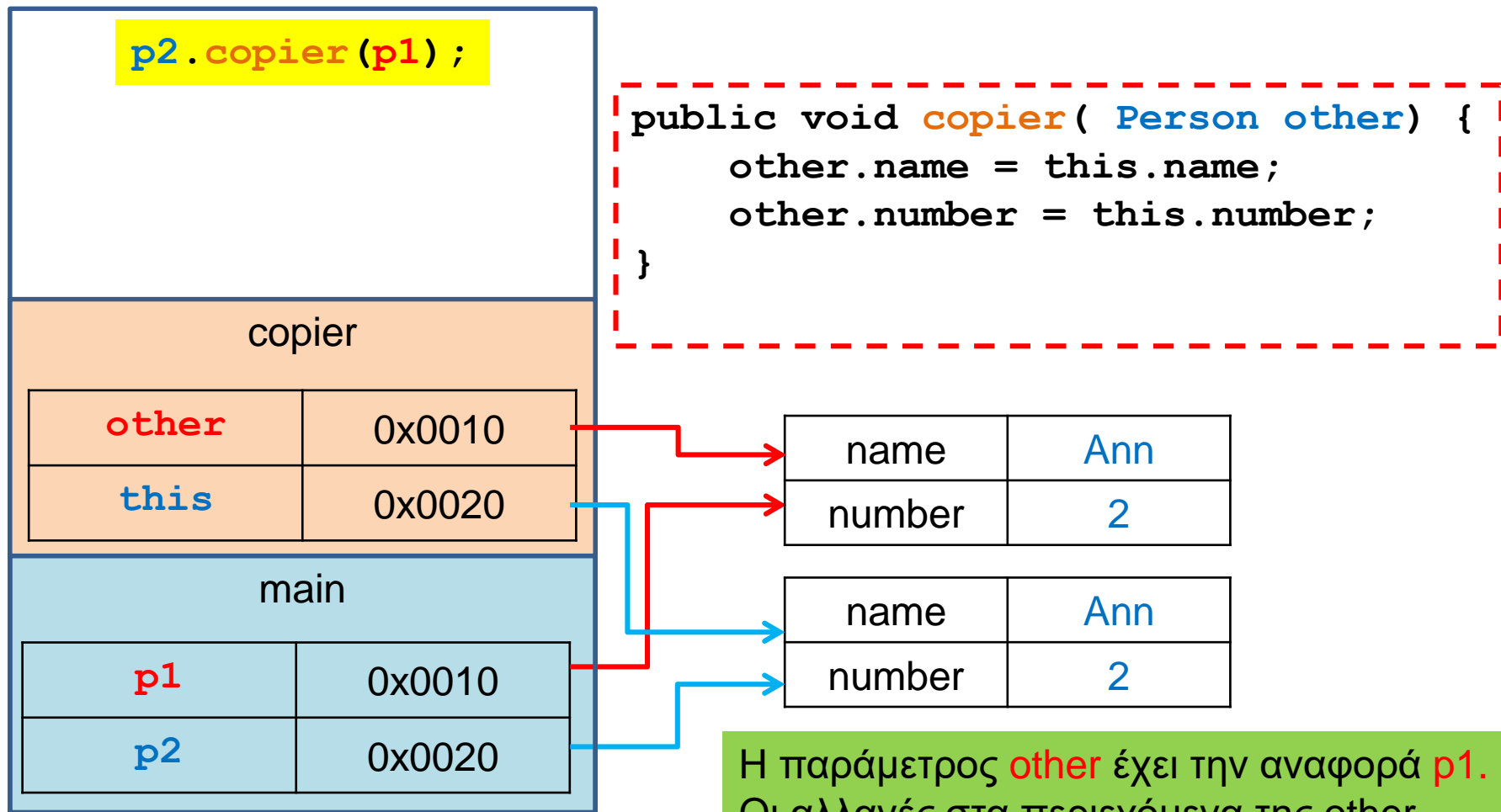
```
public void copier( Person other) {  
    other.name = this.name;  
    other.number = this.number;  
}
```

name	Bob
number	1

name	Ann
number	2

Η παράμετρος `this` περνιέται αυτόματα σε κάθε κλήση μεθόδου του αντικειμένου.
Η `other` κρατάει την αναφορά του `p1`.

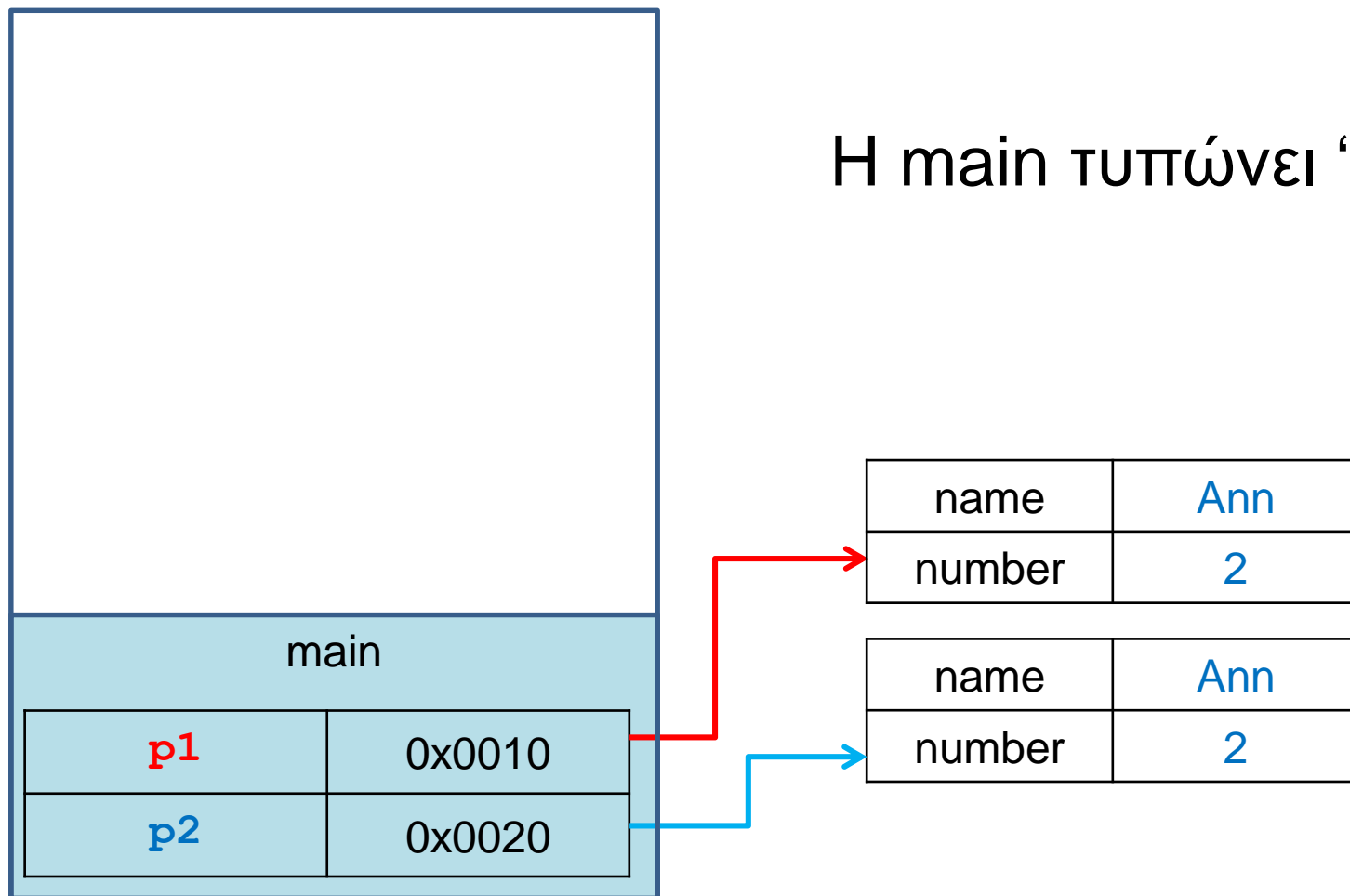
Εξέλιξη του προγράμματος



Η παράμετρος `other` έχει την αναφορά `p1`. Οι αλλαγές στα περιεχόμενα της `other` αλλάζουν και τα περιεχόμενα της `p1`.

Εξέλιξη του προγράμματος

Η main τυπώνει “Ann 2”



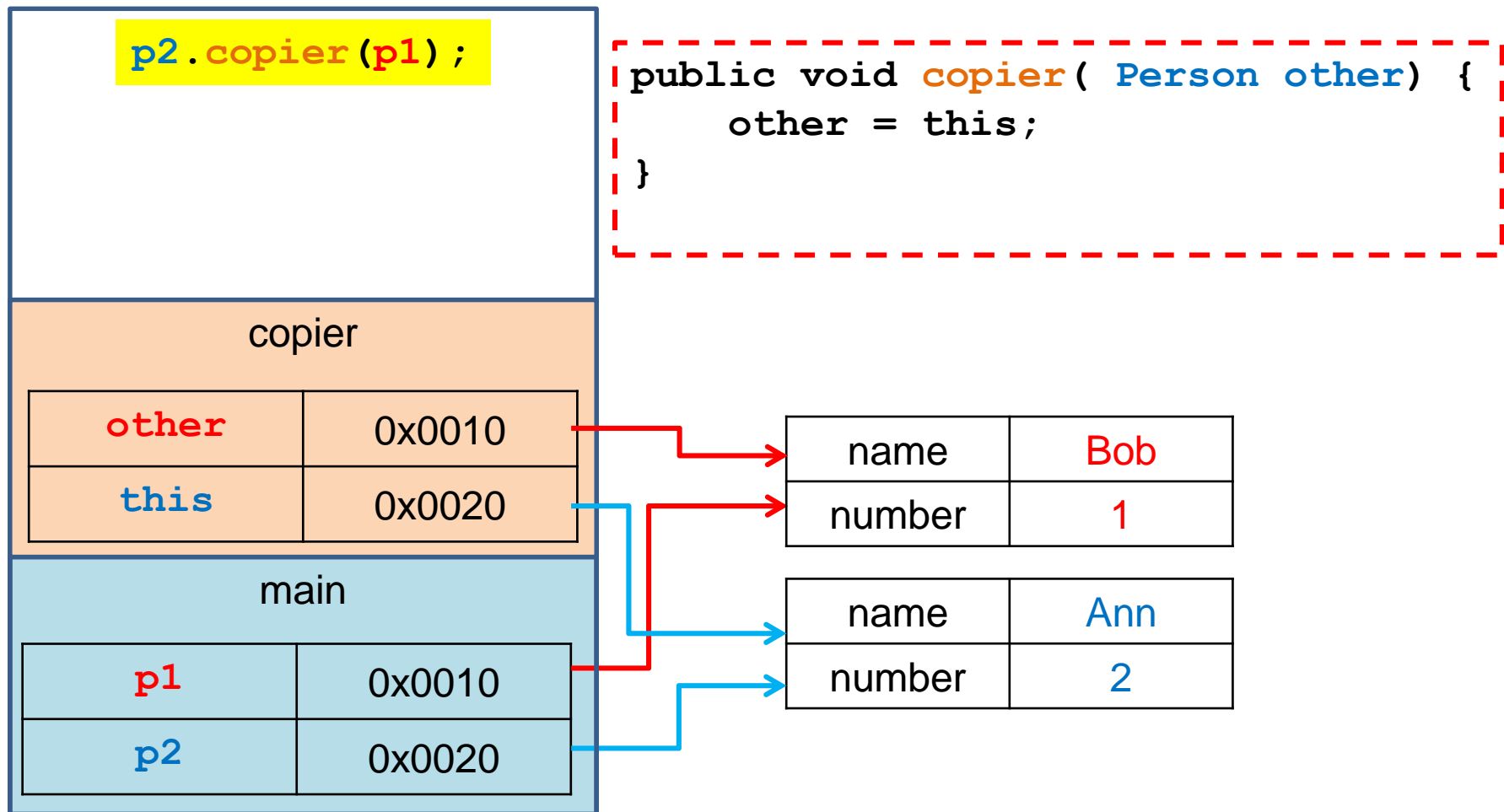
Μια άλλη υλοποίηση της copier

```
public void copier(Person other) {  
    other = this;  
}
```

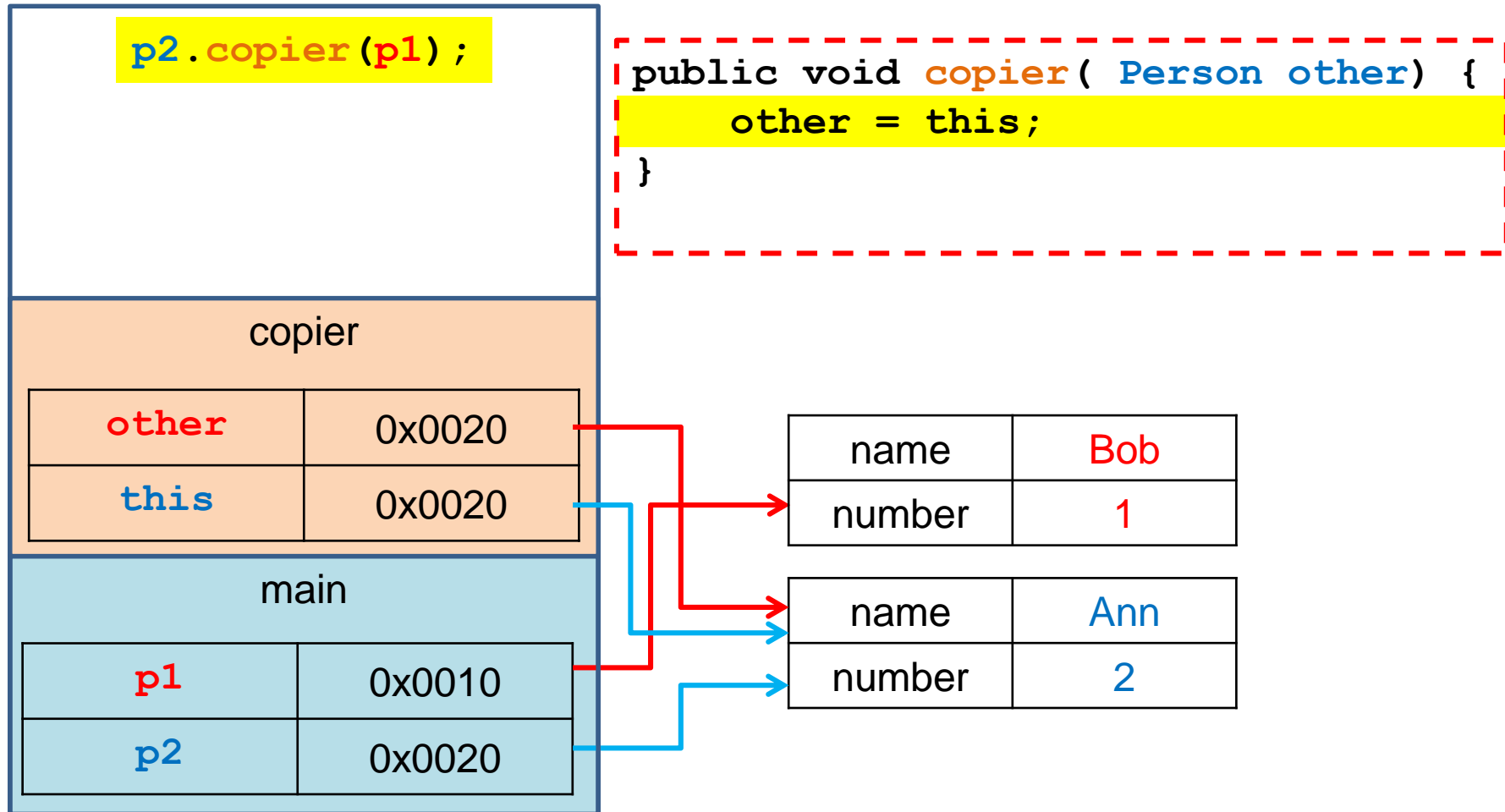
```
public class ClassParameterDemo  
{  
    public static void main(String[] args)  
    {  
        Person p1 = new Person("Bob", 1);  
        Person p2 = new Person("Ann", 2);  
        p2.copier(p1);  
        System.out.println(p1);  
    }  
}
```

Τι θα τυπώσει?

Εξέλιξη του προγράμματος

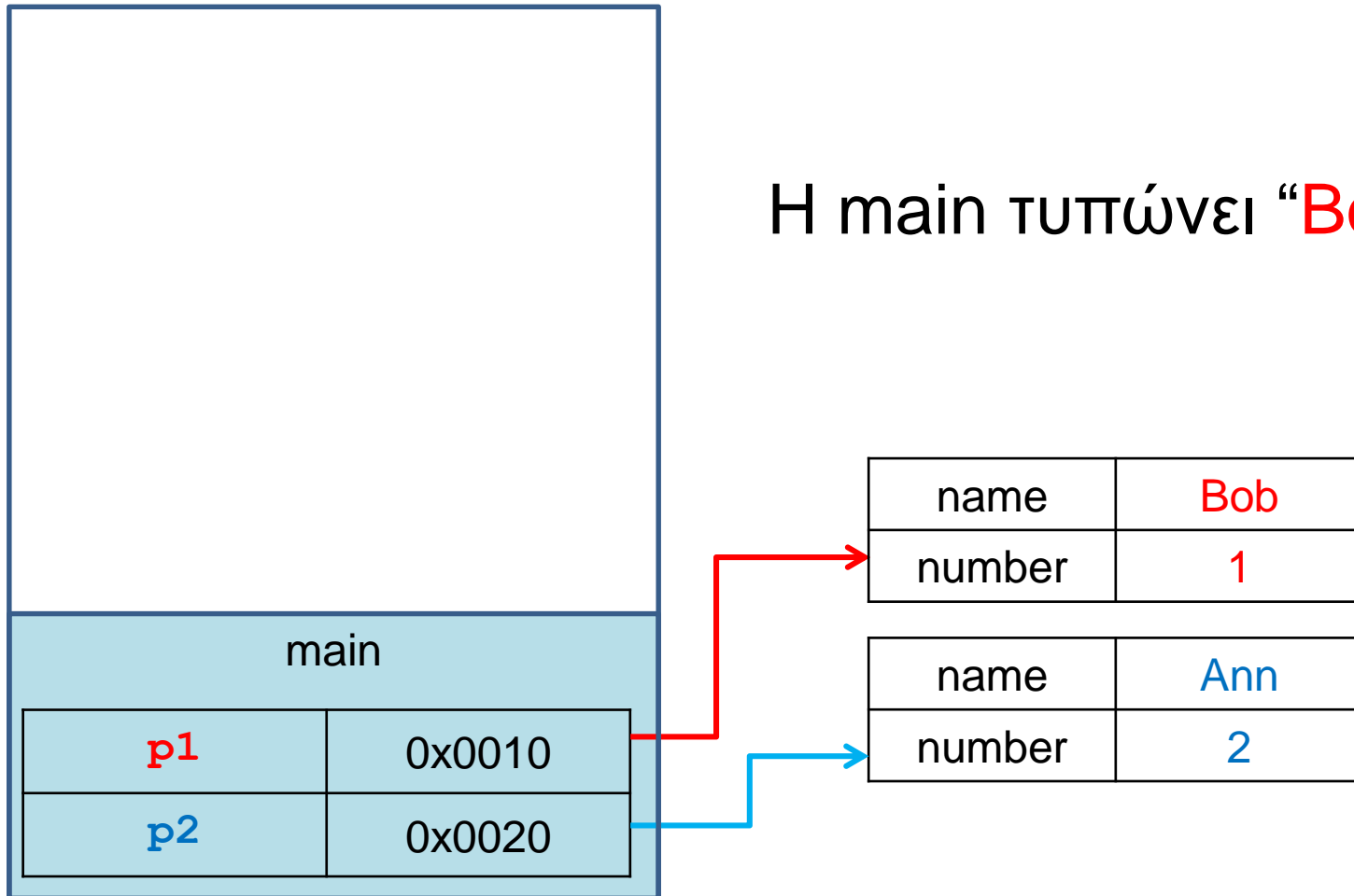


Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος

Η main τυπώνει “**Bob 1**”



Μια ακόμη υλοποίηση της copier

```
public void copier( Person other) {  
    other = new Person(this.name, this.number);  
}
```

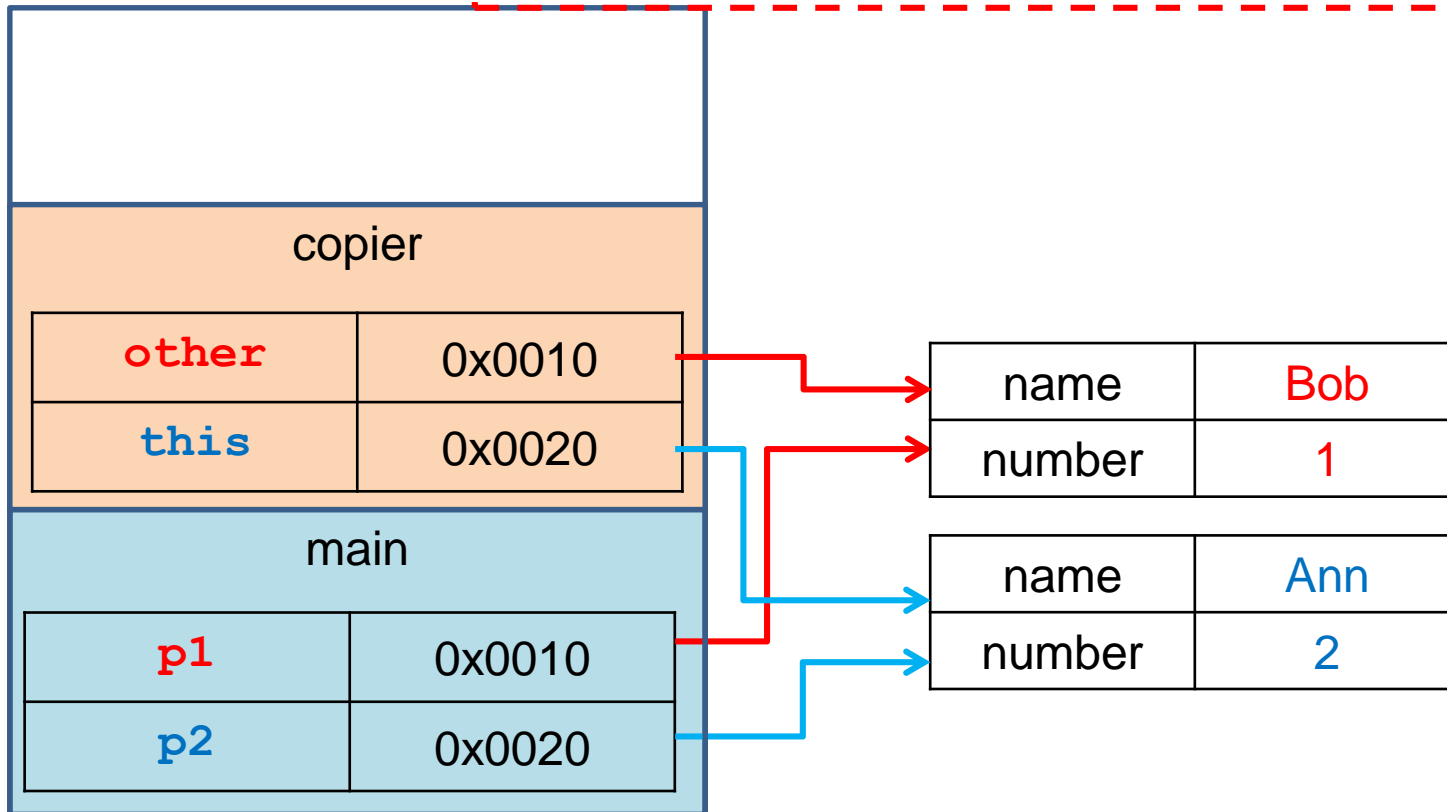
```
public class ClassParameterDemo  
{  
    public static void main(String[] args)  
    {  
        Person p1 = new Person("Bob", 1);  
        Person p2 = new Person("Ann", 2);  
        p2.copier(p1);  
        System.out.println(p1);  
    }  
}
```

Τι θα τυπώσει?

Εξέλιξη του προγράμματος

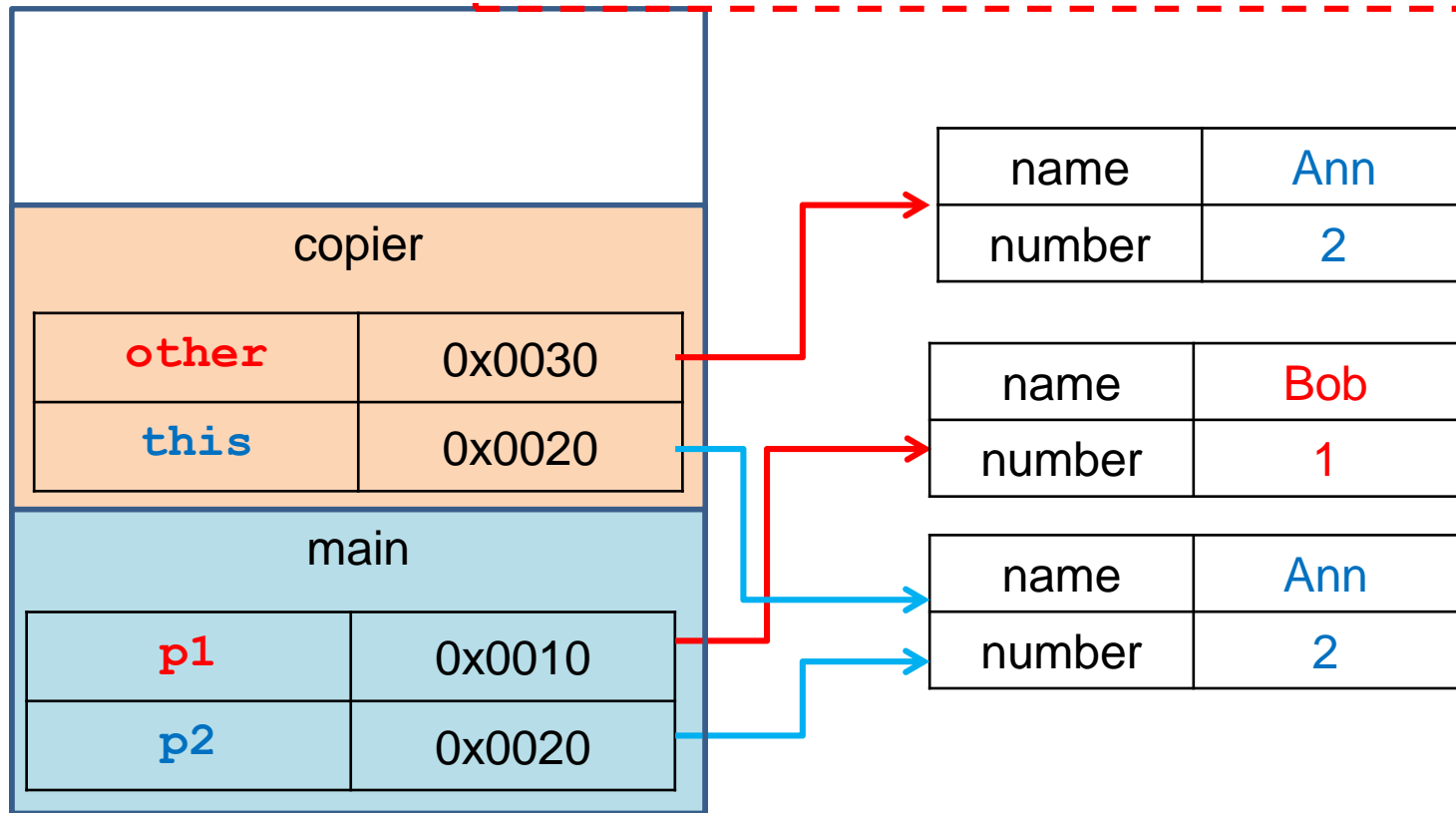
```
p2.copier(p1);
```

```
public void copier( Person other) {  
    other = new Person(this.name, this.number);  
}
```

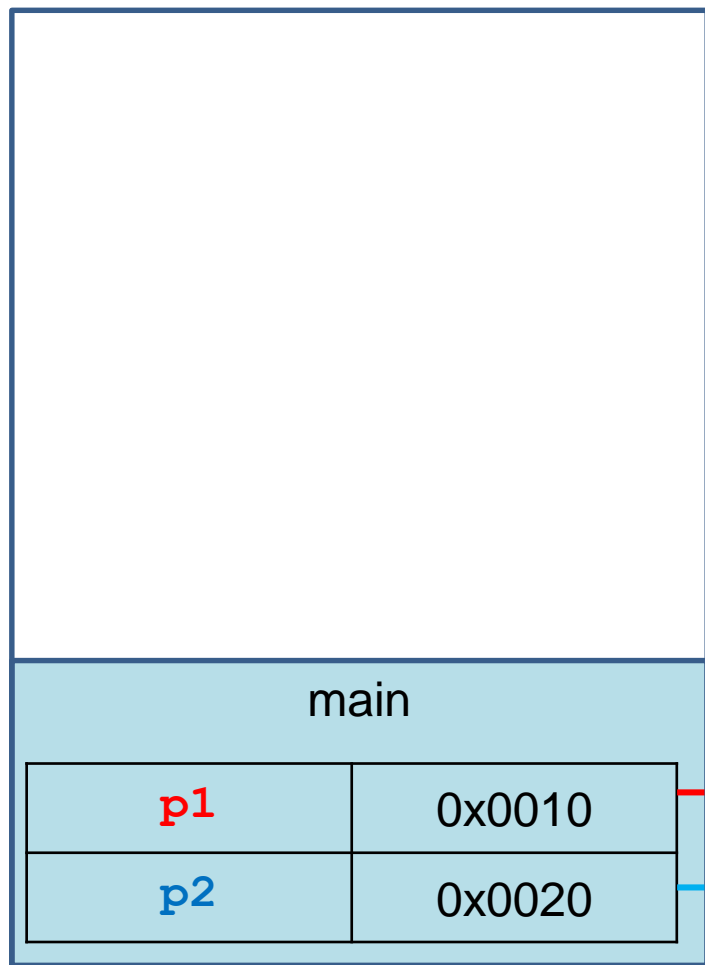


Εξέλιξη του προγράμματος

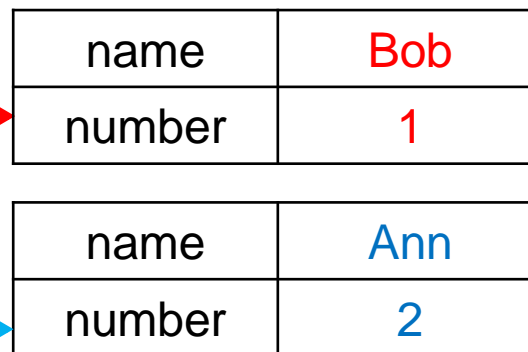
```
p2.copier(p1);  
public void copier( Person other) {  
    other = new Person(this.name, this.number);  
}
```



Εξέλιξη του προγράμματος



Η `main` τυπώνει **“Bob 1”**



Αλλαγή παραμέτρων

- Στο πρόγραμμα που είδαμε η νέα τιμή του **other** **χάνεται** όταν επιστρέφουμε από την συνάρτηση και η **p1** παραμένει αμετάβλητη.
- Αυτό γιατί το πέρασμα των παραμέτρων γίνεται κατά τιμή, και η μεταβλητή **other** είναι **τοπική**. Ότι αλλαγή κάνουμε στην τιμή της θα έχει εμβέλεια μόνο μέσα στην **copier**.
 - Το νέο αντικείμενο που δημιουργήσαμε στην περίπτωση αυτή θα χαθεί άμα φύγουμε από τη μέθοδο εφόσον δεν υπάρχει κάποια αναφορά σε αυτό.
- Η αλλαγή στην **τιμή** της **other** είναι διαφορετική από την αλλαγή στα **περιεχόμενα** της διεύθυνσης στην οποία δείχνει η **other**
 - Οι αλλαγές στα περιεχόμενα αλλάζουν τον χώρο μνήμης στο σωρό (heap). Οι αλλαγές επηρεάζουν όλες τις αναφορές στο αντικείμενο.

```
class ArrayVar{
    public static void main(String[] args){
        int[] array = {1,2,3};
        int x = 5;

        increment(array);
        for (int i = 0; i < array.length; i ++){ System.out.print(array[i] + " "); }

        increment(x);
        System.out.println("x: " + x);

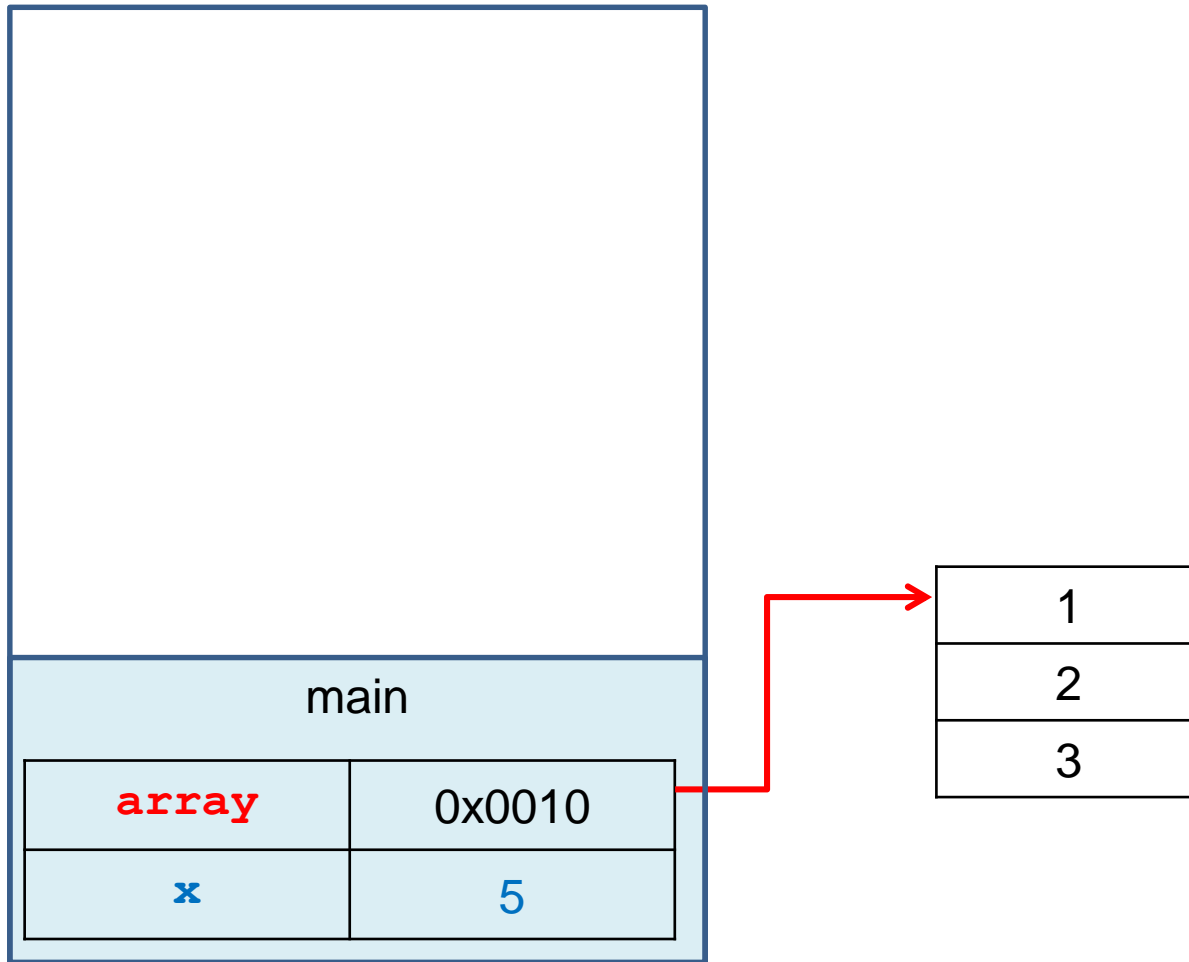
        increment(array[0]);
        System.out.println("array[0] = " + array[0]);
    }

    public static void increment(int[] array){
        for (int i = 0; i < array.length; i ++){
            array[i] ++;
            System.out.print(array[i] + " ");
        }
    }

    public static void increment(int x){
        x ++ ;
        System.out.println("x: " + x);
    }
}
```

Τι θα τυπώσει?

Πέρασμα παραμέτρων



Πέρασμα παραμέτρων

`increment(array)`

increment

`array`

0x0010

main

`array`

0x0010

`x`

5

```
public static void increment(int[] array){  
    for (int i = 0; i < array.length; i ++){  
        array[i] ++;  
        System.out.print(array[i] + " ");  
    }  
    System.out.println("");  
}
```

1

2

3

Πέρασμα παραμέτρων

increment(array)

```
public static void increment(int[] array){  
    for (int i = 0; i < array.length; i ++){  
        array[i] ++;  
        System.out.print(array[i] + " ");  
    }  
    System.out.println("");  
}
```

increment

array

0x0010

main

array

0x0010

x

5

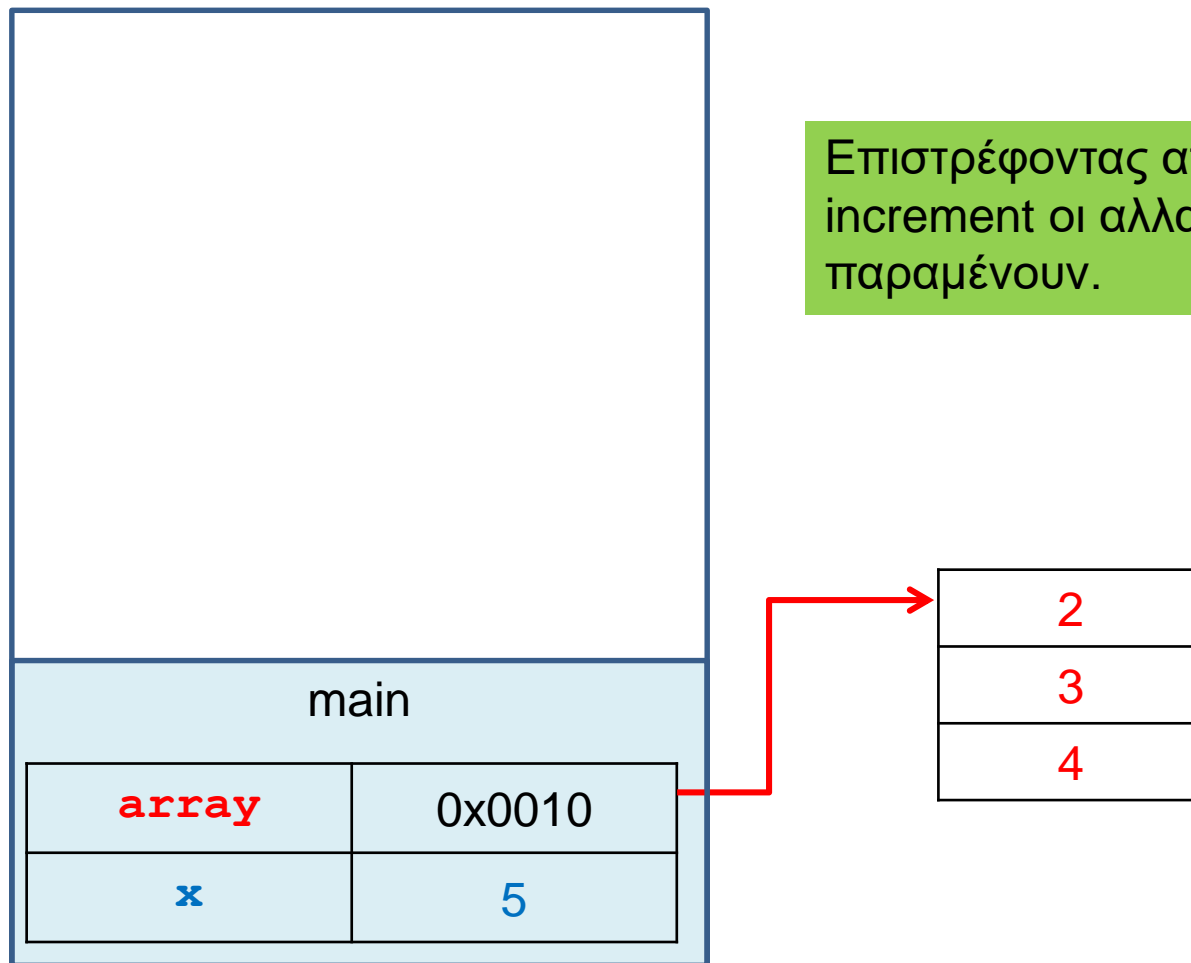
2

3

4

Πέρασμα παραμέτρων

Επιστρέφοντας από την μέθοδο `increment` οι αλλαγές στον πίνακα παραμένουν.



Πέρασμα παραμέτρων

increment(x)

```
public static void increment(int x) {  
    x ++;  
    System.out.println("x: " + x);  
}
```

increment

x

5

main

array

0x0010

x

5

2

3

4

Πέρασμα παραμέτρων

increment(x)

increment

x

6

main

array

0x0010

x

5

```
public static void increment(int x) {  
    x ++;  
    System.out.println("x: " + x);  
}
```

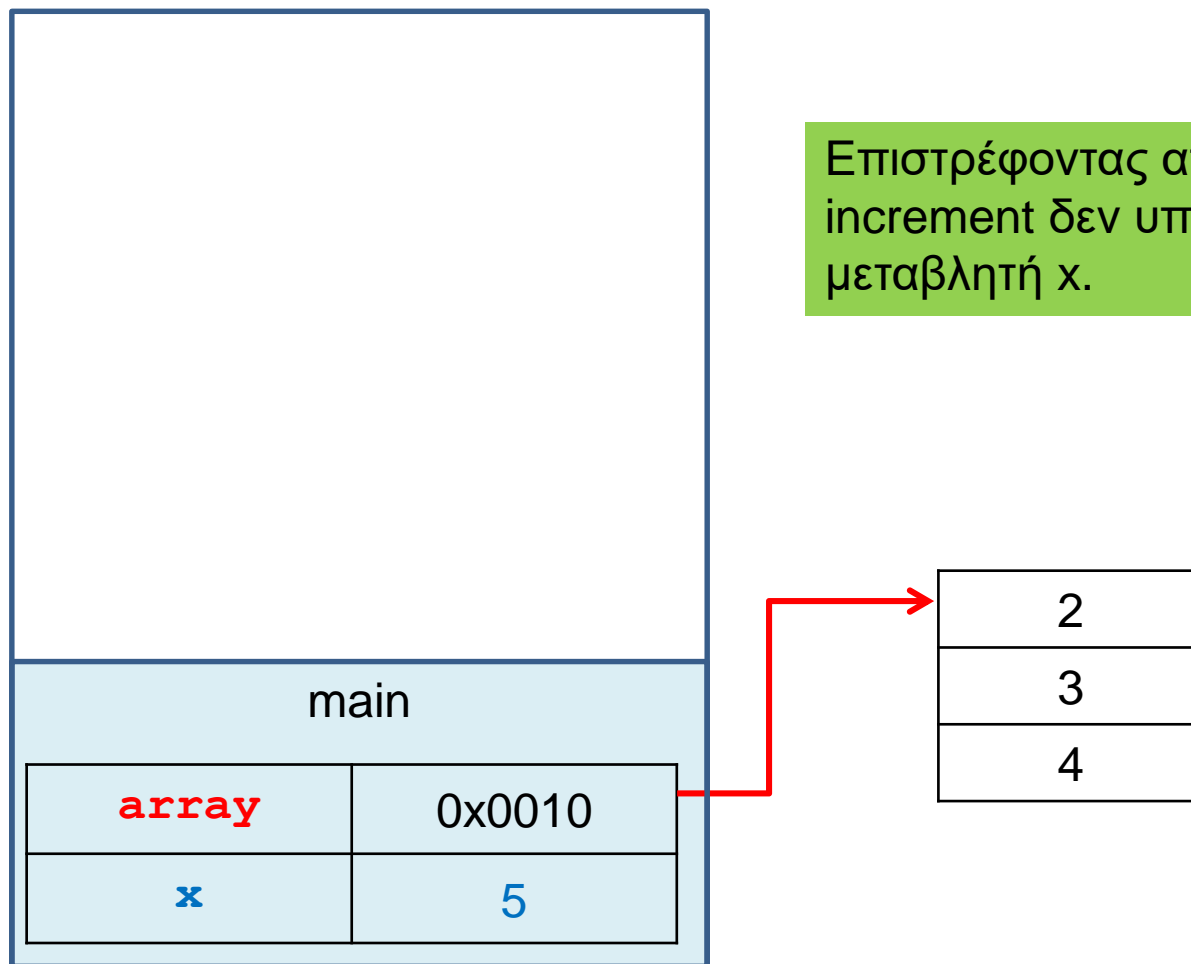
2

3

4

Πέρασμα παραμέτρων

Επιστρέφοντας από την μέθοδο `increment` δεν υπάρχουν αλλαγές στη μεταβλητή `x`.



Πέρασμα παραμέτρων

`increment(array[0])`

```
public static void increment(int x) {  
    x ++;  
    System.out.println("x: " + x);  
}
```

increment

`x`

2

main

`array`

0x0010

`x`

5

2

3

4

Πέρασμα παραμέτρων

increment(array[0])

```
public static void increment(int x) {  
    x ++;  
    System.out.println("x: " + x);  
}
```

increment

x

3

main

array

0x0010

x

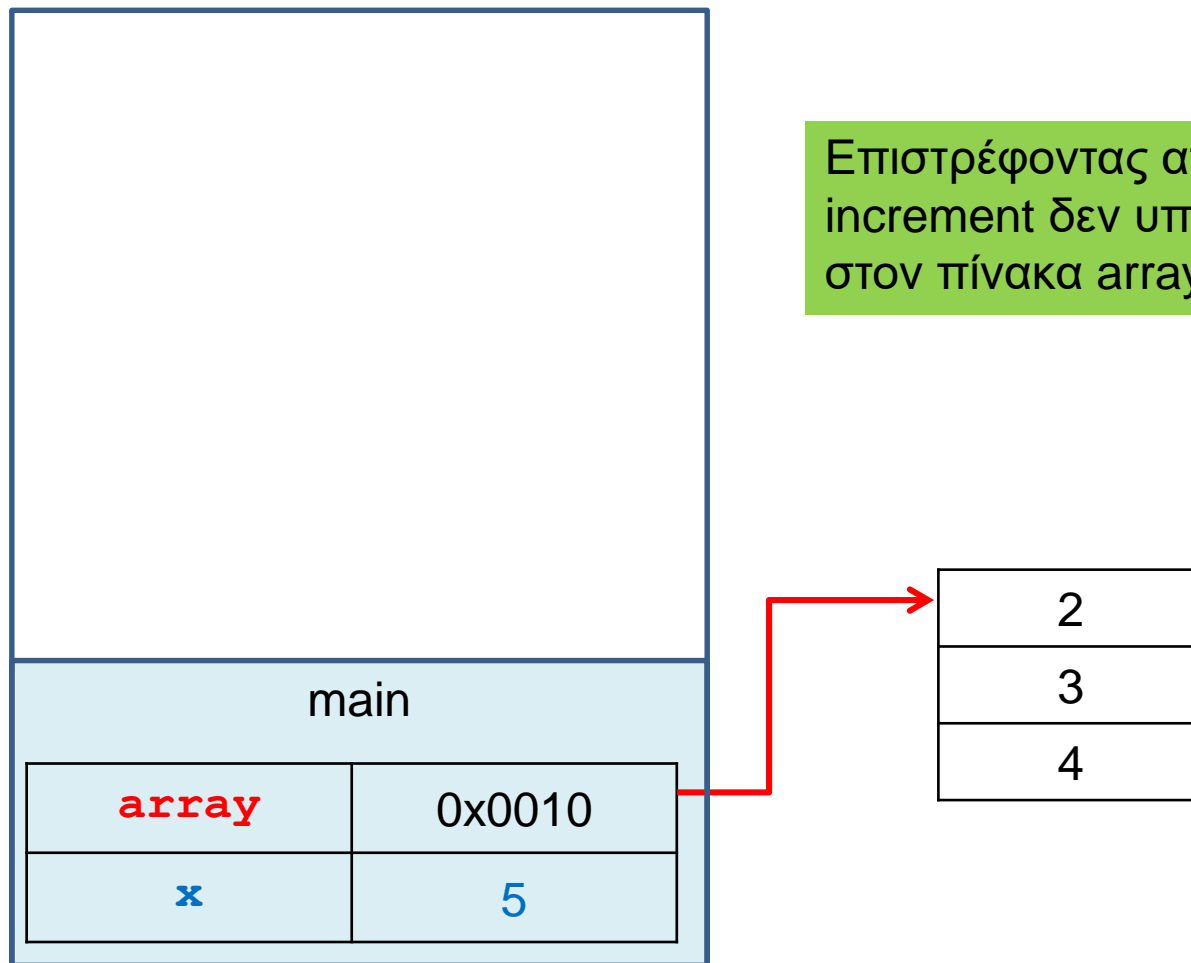
5

2

3

4

Πέρασμα παραμέτρων



Επιστρέφοντας από την μέθοδο `increment` δεν υπάρχουν αλλαγές στη στον πίνακα `array`.

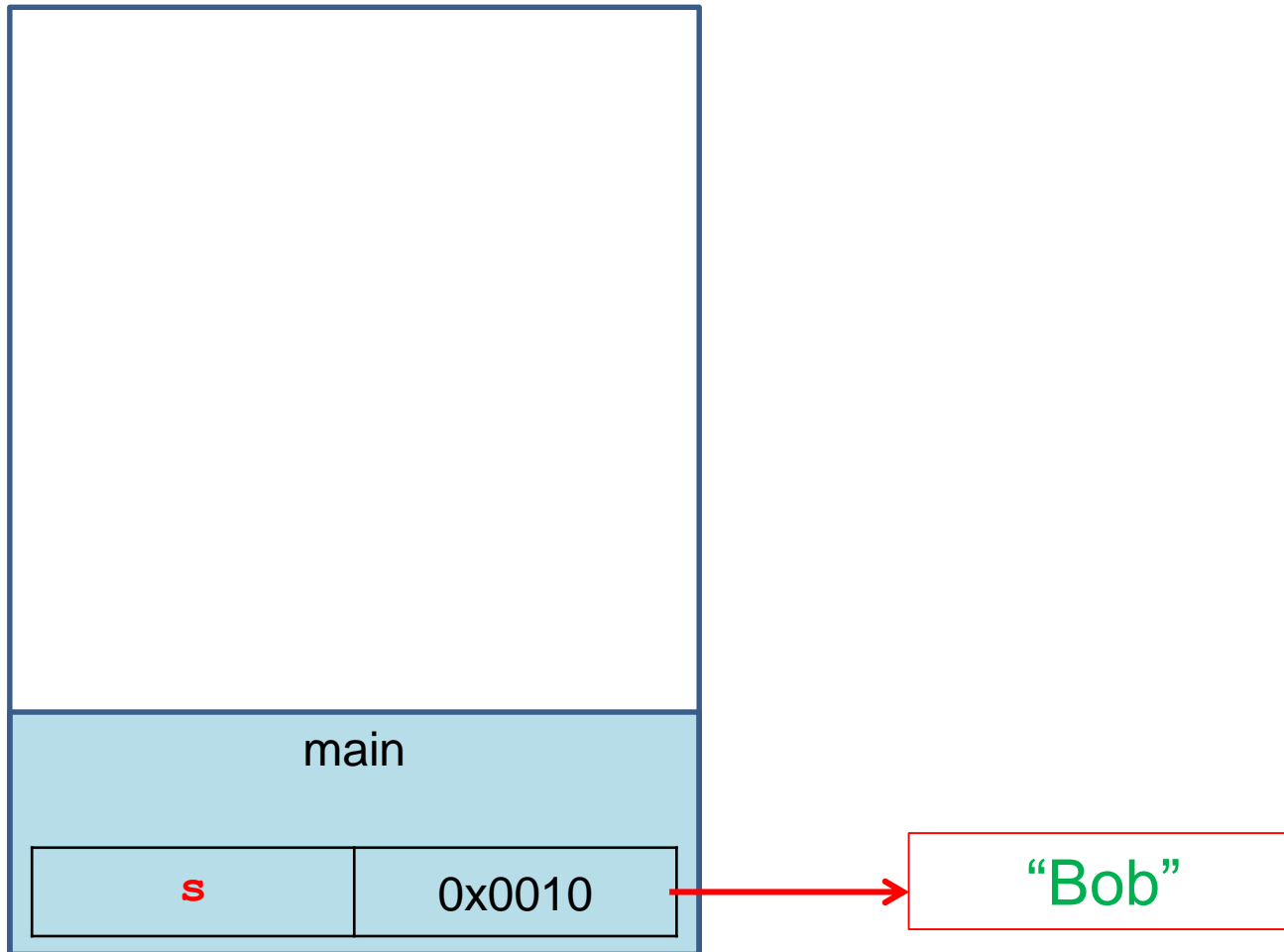
Άλλο ένα παράδειγμα

```
public class StringParameterDemo
{
    public static void main(String[] args)
    {
        String s = "Bob";
        changeString(s);
        System.out.println(s);
    }

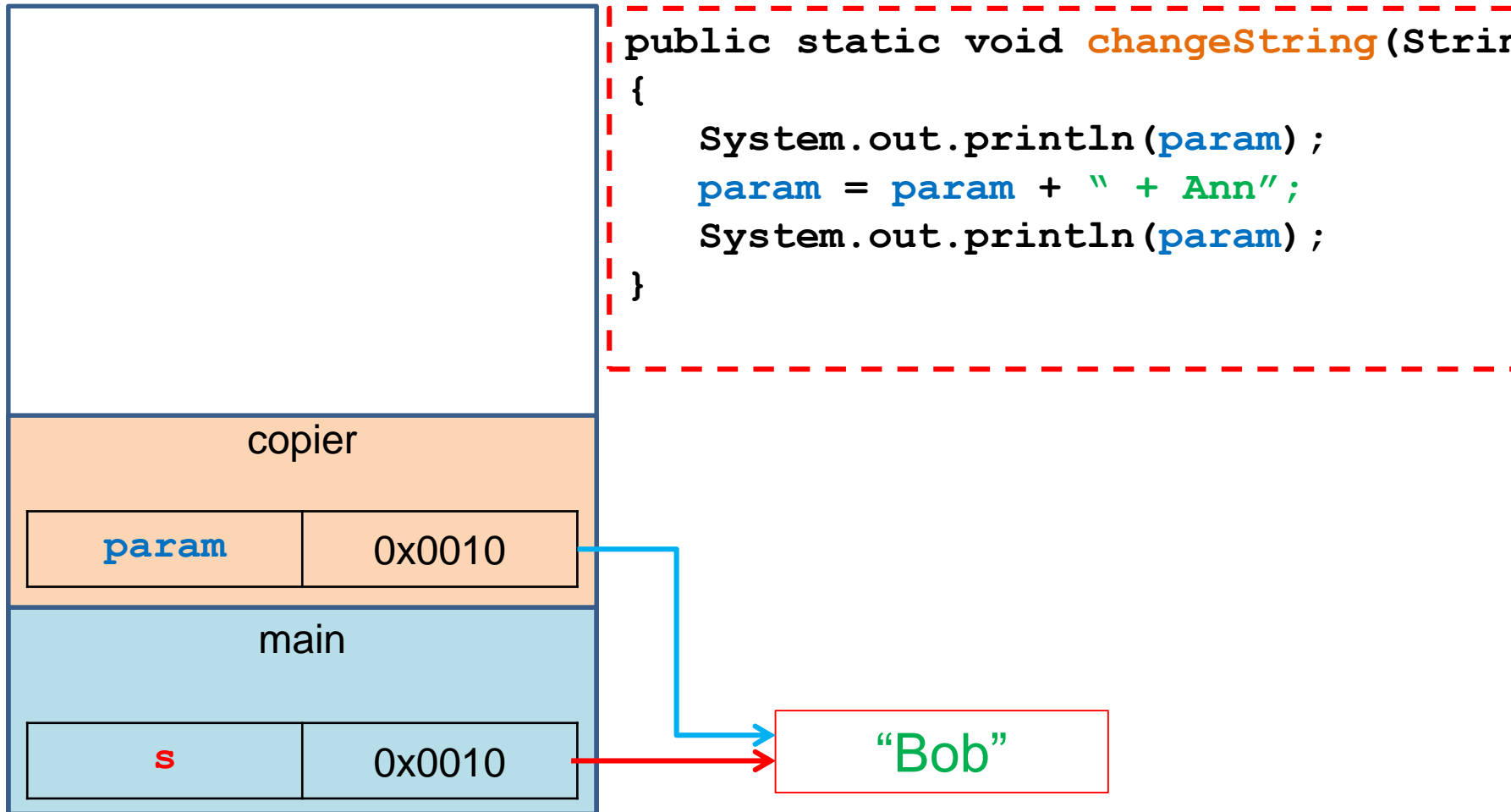
    public static void changeString(String param)
    {
        System.out.println(param);
        param = param + " + Ann";
        System.out.println(param);
    }
}
```

Τι θα τυπώσει?

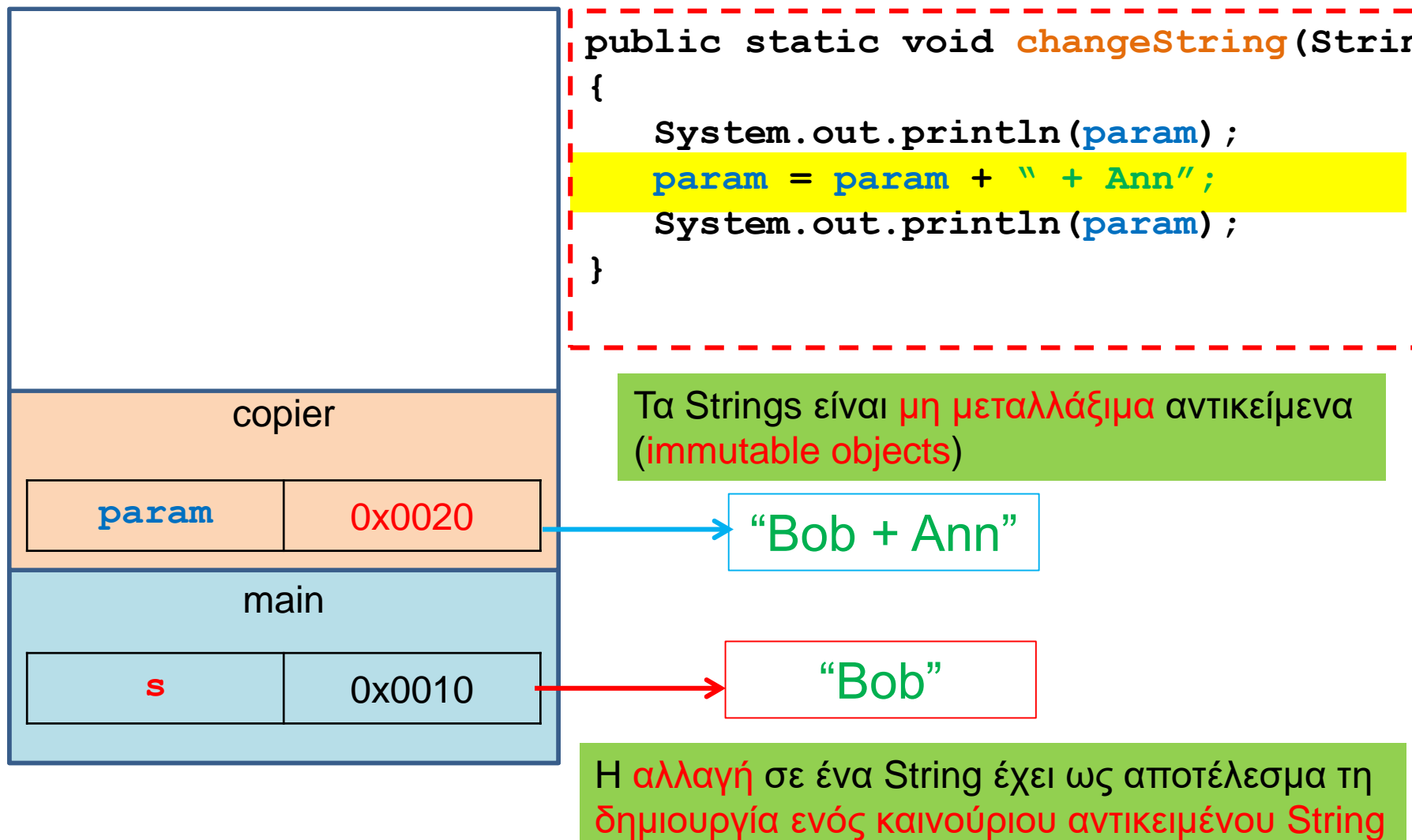
Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος



ΕΛΕΓΧΟΣ ΙΣΟΤΗΤΑΣ

Έλεγχος ισότητας

- Έχουμε πει ότι όταν ελέγχουμε ισότητα μεταξύ αντικειμένων (π.χ., Strings) πρέπει να γίνεται μέσω της μεθόδου **equals** και όχι με το **==**
- Η συζήτηση με τις αναφορές εξηγεί γιατί η σύγκριση με **==** δε δουλεύει
- Η σύγκριση με **==** συγκρίνει αν δύο **αναφορές** είναι ίδιες και **όχι** αν **τα περιεχόμενα** των θέσεων μνήμης στις οποίες δείχνουν οι αναφορές είναι ίδια.

```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber){
        name = initName;
        number = initNumber;
    }

    public void set(String newName, int newNumber){
        name = newName;
        number = newNumber;
    }

    public boolean equals(Person other){
        return this.name.equals(other.name) && this.number == other.number;
    }

    public void copier(Person other) {
        other.name = name;
        other.number = number;
    }

    public String toString( ){
        return (name + " " + number);
    }
}
```

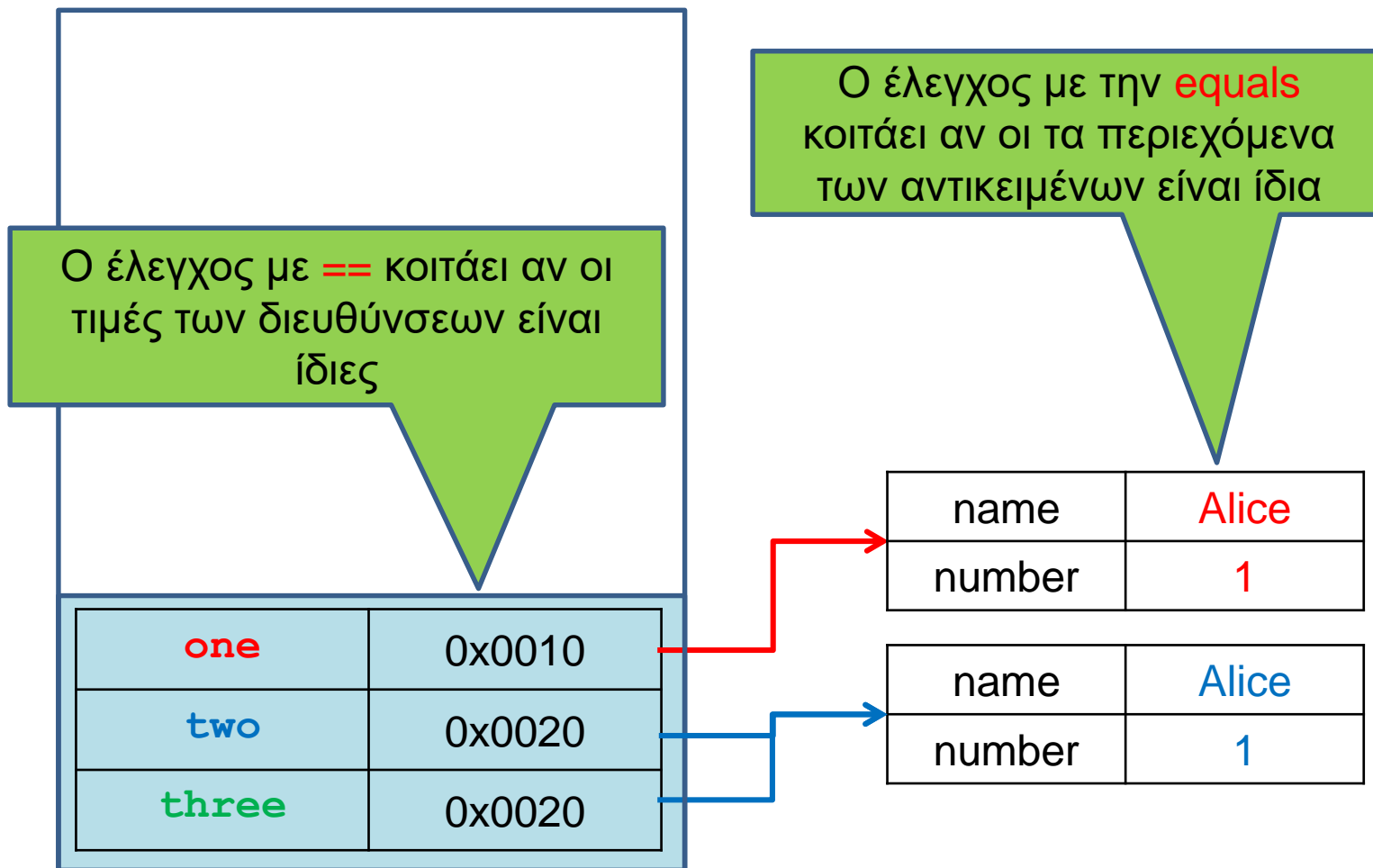
Παράδειγμα

- Τι θα τυπώσει ο παρακάτω κώδικας?

```
Person one = new Person("Alice", 1);
Person two = new Person("Alice", 1);
Person three = two;
System.out.println(one == two);
System.out.println(two == three);
System.out.println(one == three);
System.out.println(one.equals(two));
System.out.println(two.equals(three));
System.out.println(one.equals(three));
```

false
true
false
true
true
true

Εξήγηση



STRING INTERNING

String Interning

- Στην Java για κάθε **string value** (σταθερά) που εμφανίζεται δημιουργείται ένα **αντικείμενο**, το οποίο ονομάζεται **intern string**, και το οποίο κρατάει αυτή την τιμή.
- Για αυτό και οι αλφαριθμητικές σταθερές μπορούν να χρησιμοποιηθούν και σαν αντικείμενα. Π.χ. μπορούμε να καλέσουμε:

```
"java".length()
```

Καλούμε μια μέθοδο του intern String

- Αυτό μπορεί να προκαλέσει μπερδέματα με ελέγχους ισότητας.

Ισότητα String

Τι θα εκτυπωθεί?

```
import java.util.Scanner;

class StringEquality{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);

        String x = input.next();
        String z = new String("java");
        String y = "java";

        System.out.println("1. " + (x == "java"));
        System.out.println("2. " + (y == "java"));
        System.out.println("3. " + (z == "java"));
        System.out.println("4. " + x.equals("java"));
        System.out.println("5. " + y.equals("java"));
        System.out.println("6. " + z.equals("java"));
    }
}
```

1. false

2. true

3. false

4. true

5. true

6. true

Για την σύγκριση Strings **ΠΑΝΤΑ** χρησιμοποιούμε την μέθοδο **equals**.

String Interning

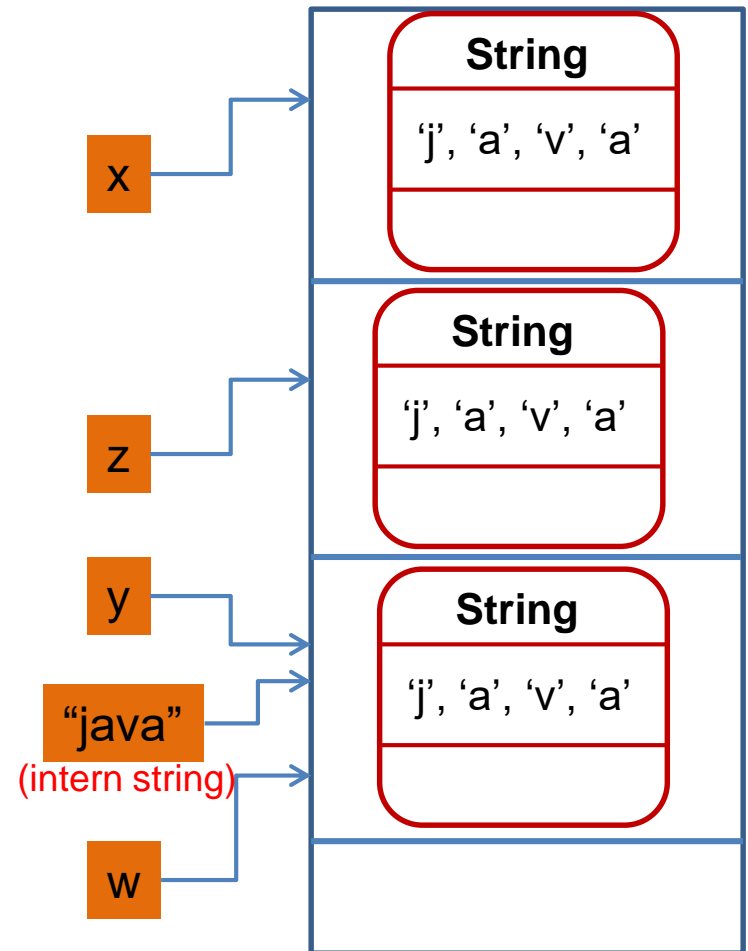
- Όταν γίνεται η εκχώρηση της σταθεράς `"java"` δημιουργείται ένα **intern string**, και το οποίο κρατάει αυτή την σταθερά.

- Η εντολή

```
String y = "java";
```

κάνει το `y` να δείχνει στη θέση που είναι αποθηκευμένη η σταθερά `"java"`

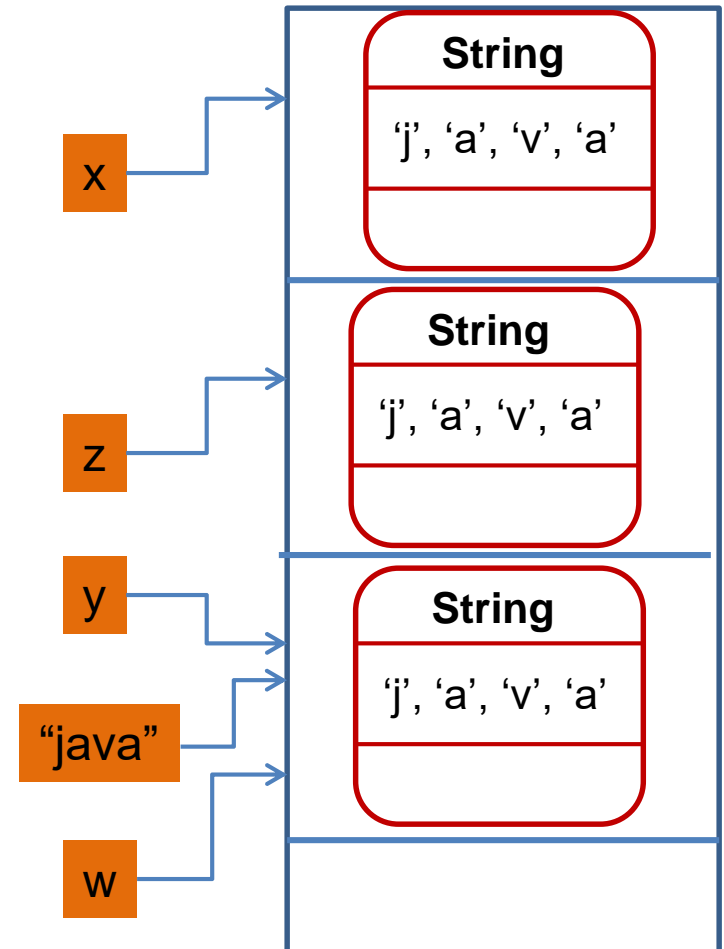
```
String x = input.next();  
String z = new String("java");  
String y = "java";  
String w = "java";
```



String Interning

```
String x = input.next();  
String z = new String("java");  
String y = "java";  
String w = "java";  
System.out.println((y == "java"));
```

- Ο τελεστής `==` μεταξύ δύο αντικειμένων εξετάζει αν πρόκειται για την **ίδια θέση μνήμης**.
- Γι αυτό (`y == "java"`) επιστρέφει **true** ενώ το (`z == "java"`) επιστρέφει **false**.



```
class StringClass
```

```
{  
    String s = "abc";
```

```
    public void changeObject(StringClass other) {
```

```
        if (this.s == other.s) {  
            System.out.println("Same");
```

```
        }else {  
            System.out.println("Different");
```

```
        }
```

```
        String local = new String("local");
```

```
        other.s = local;
```

```
        local = "local";
```

```
        s = local;
```

```
        if (this.s == other.s) {  
            System.out.println("Same");
```

```
        }else {  
            System.out.println("Different");
```

```
        }
```

```
    }
```

```
}
```

Η ανάθεση String σταθεράς έχει αποτέλεσμα την δημιουργία ενός intern string στο οποίο δείχνουν όλα τα strings στα οποία ανατίθεται η σταθερά.

Η ανάθεση String σταθεράς είναι διαφορετική από τη δημιουργία αντικειμένου με new

Η σταθερά δημιουργεί ένα νέο **intern String**

Τι θα τυπώσει?

```
class StringTest{
```

```
    public static void main(String[] args) {
```

```
        StringClass obj1 = new StringClass();
```

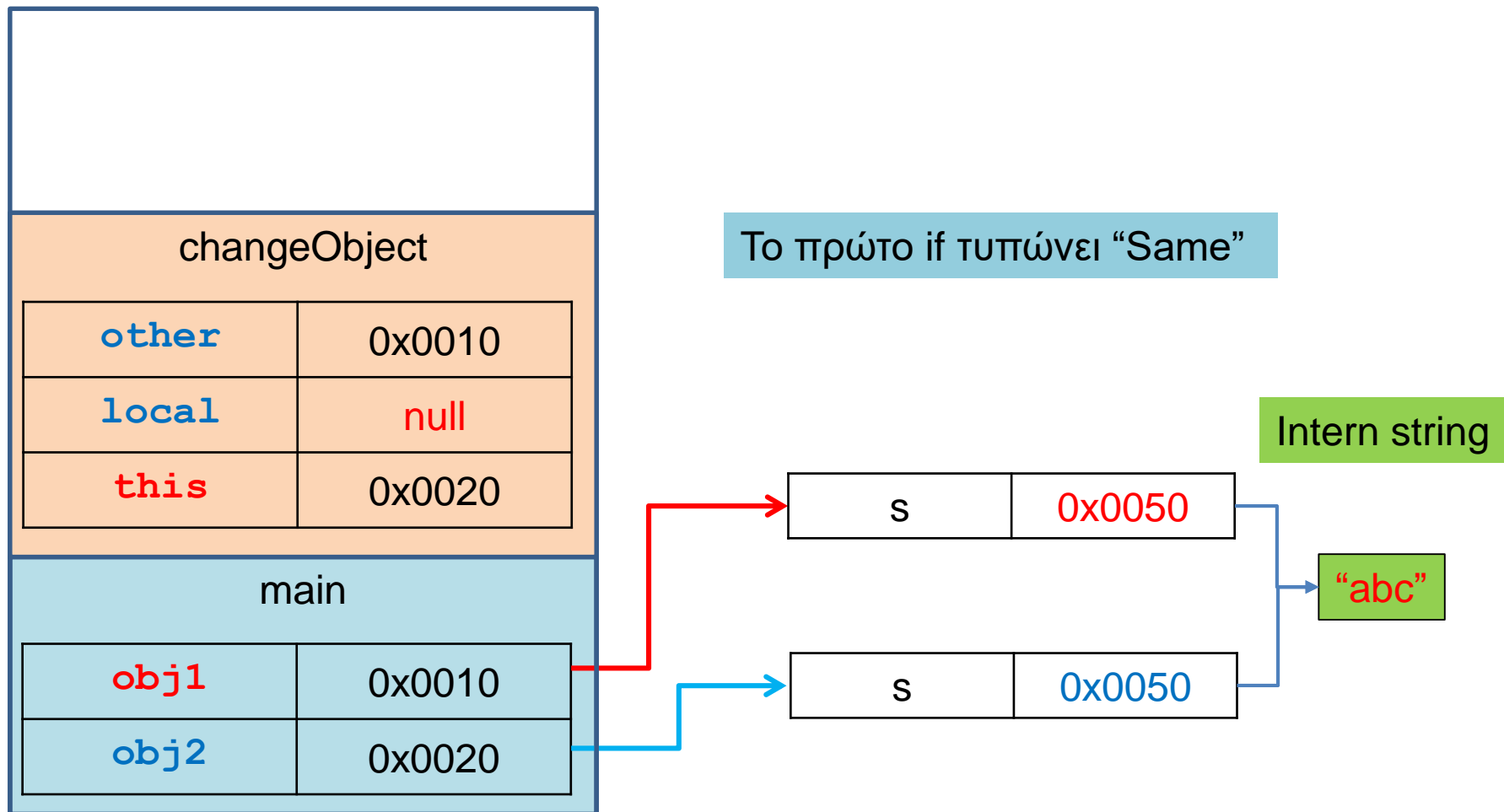
```
        StringClass obj2 = new StringClass();
```

```
        obj2.changeObject(obj1);
```

```
    }
```

```
}
```

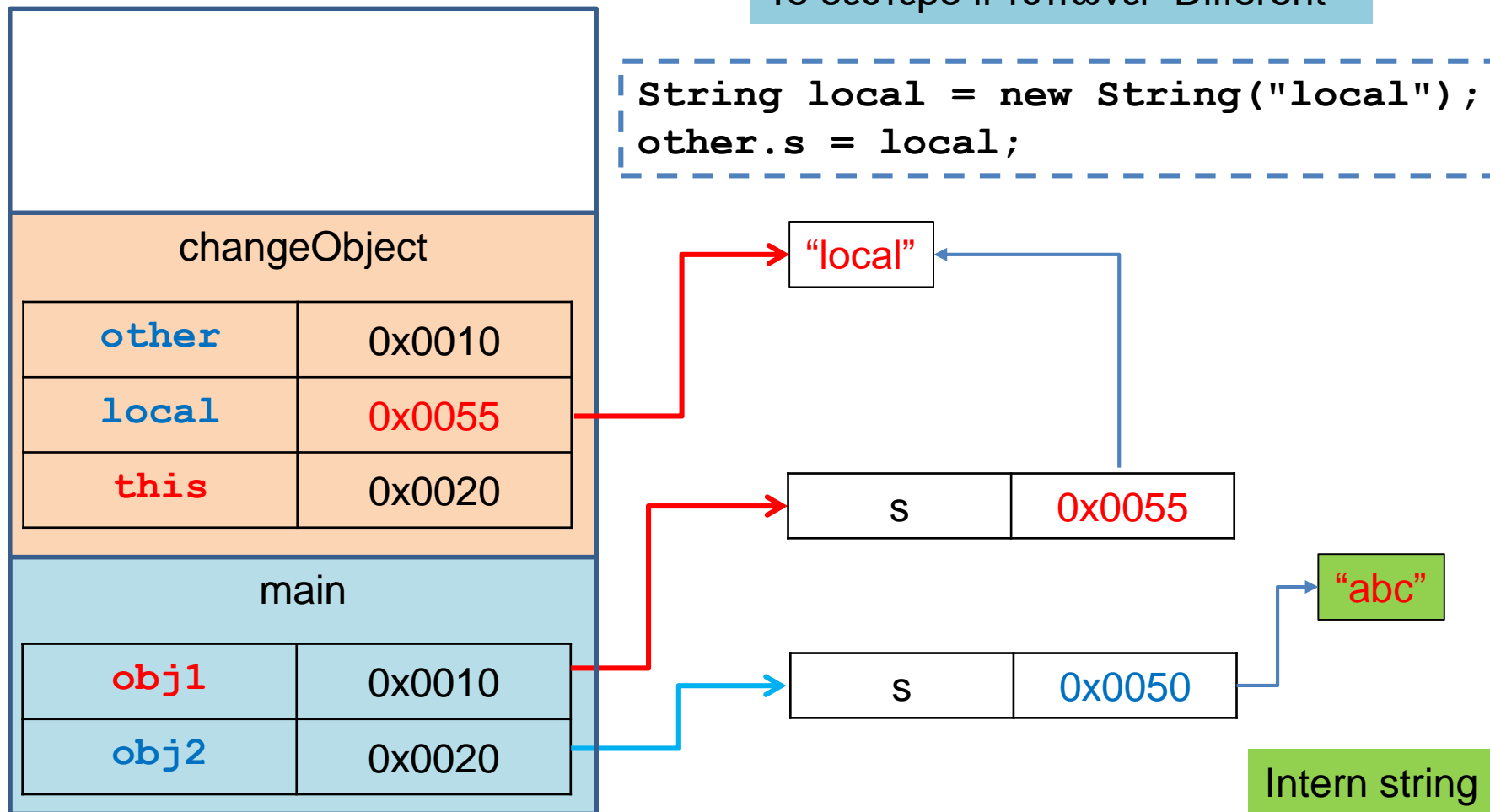
Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος

Το δεύτερο if τυπώνει "Different"

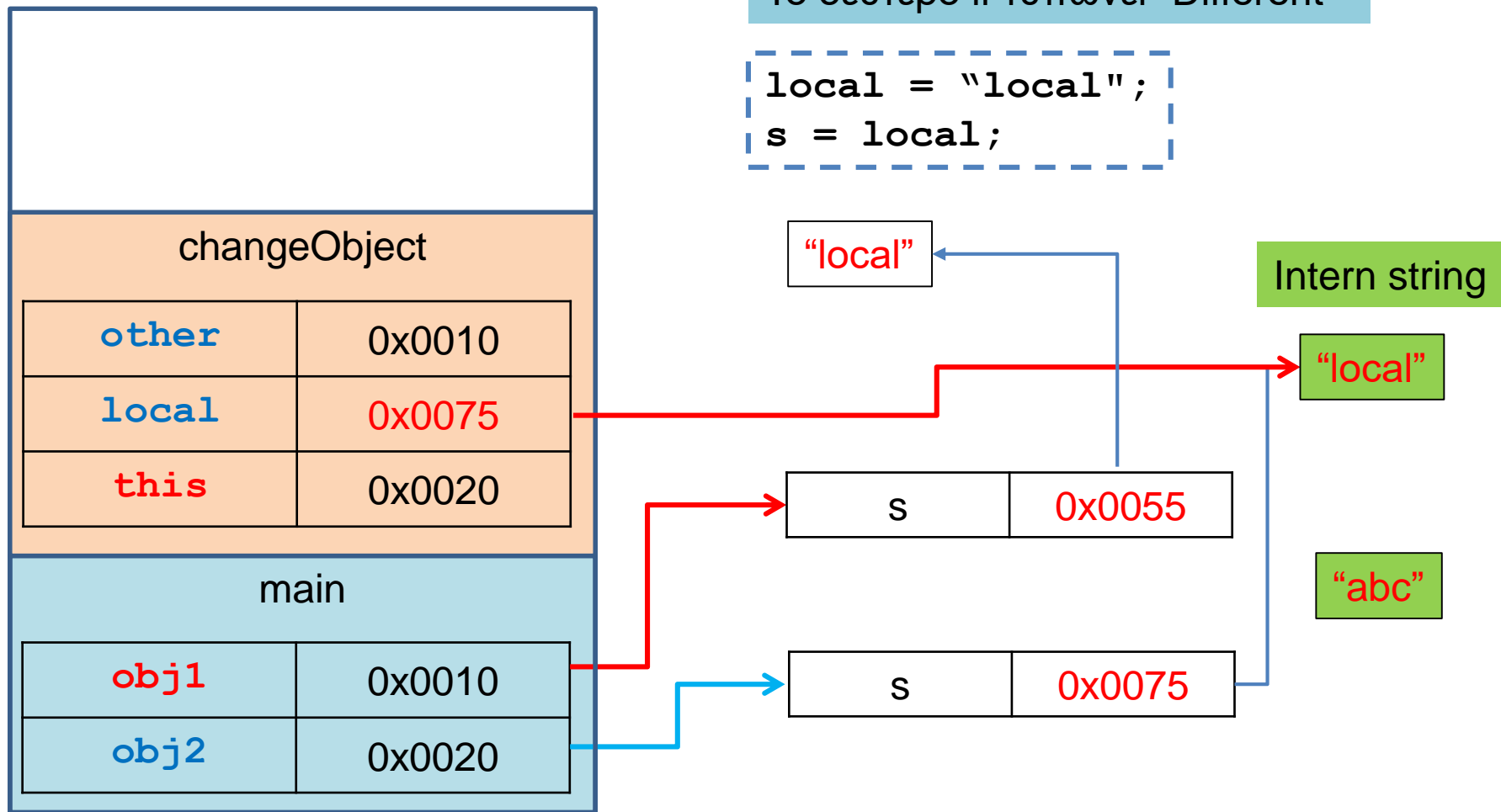
```
String local = new String("local");  
other.s = local;
```



Εξέλιξη του προγράμματος

Το δεύτερο if τυπώνει "Different"

```
local = "local";  
s = local;
```



ΑΠΟ-ΑΝΑΦΟΡΟΠΟΙΗΣΗ (DEREFERENCING)

```
class Person
{
    private String name;

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

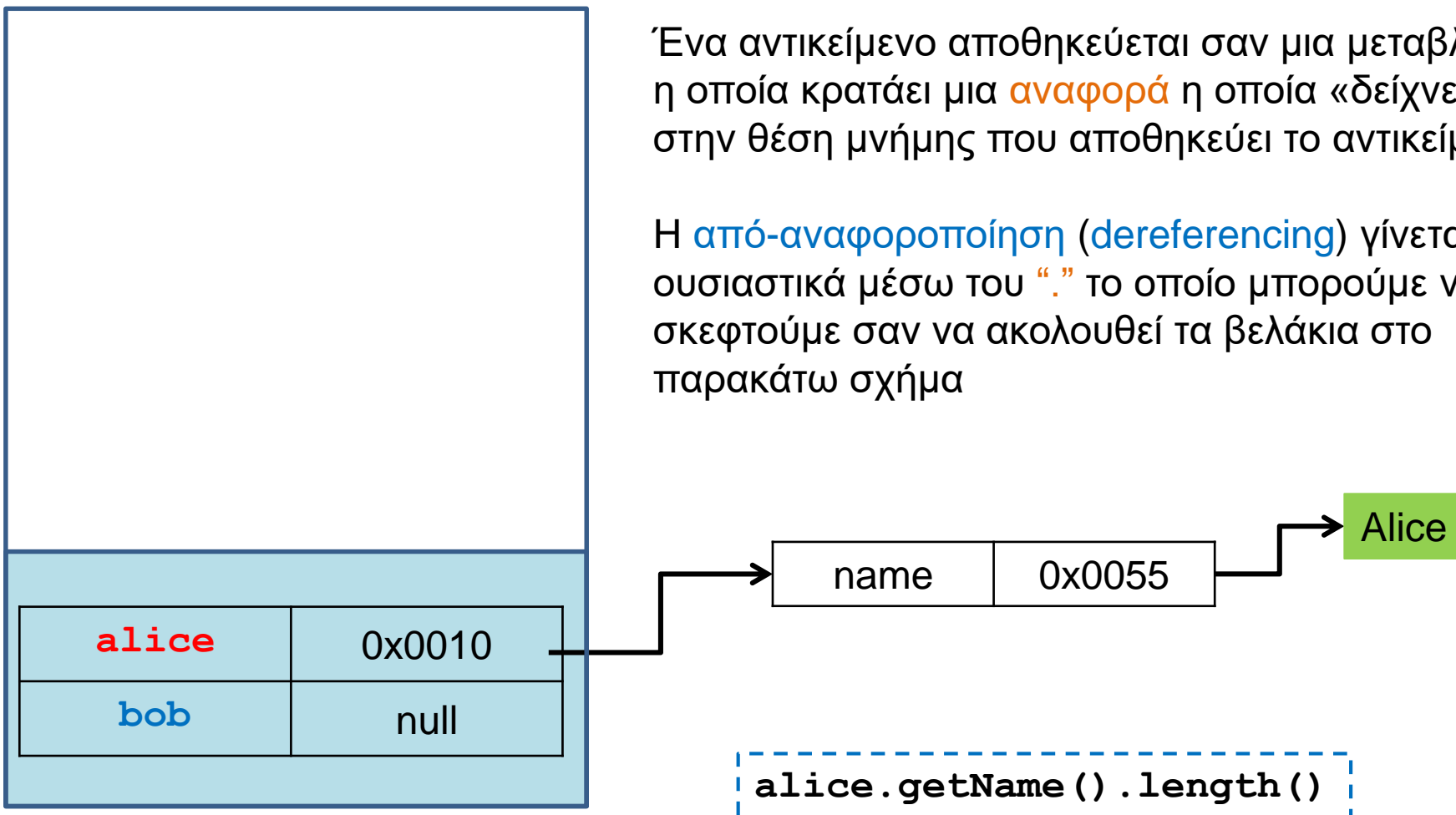
    public String getName() {
        return name;
    }
}
```

```
class PersonTest
{
    public static void main(String args[])
    {
        Person alice = new Person("Alice");
        Person bob;
        System.out.println(alice.getName());
        System.out.println(alice.getName().length());
    }
}
```

Dereferencing

Ένα αντικείμενο αποθηκεύεται σαν μια μεταβλητή η οποία κρατάει μια αναφορά η οποία «δείχνει» στην θέση μνήμης που αποθηκεύει το αντικείμενο.

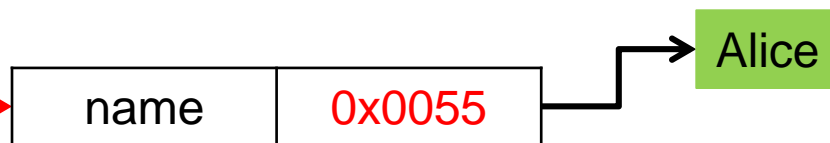
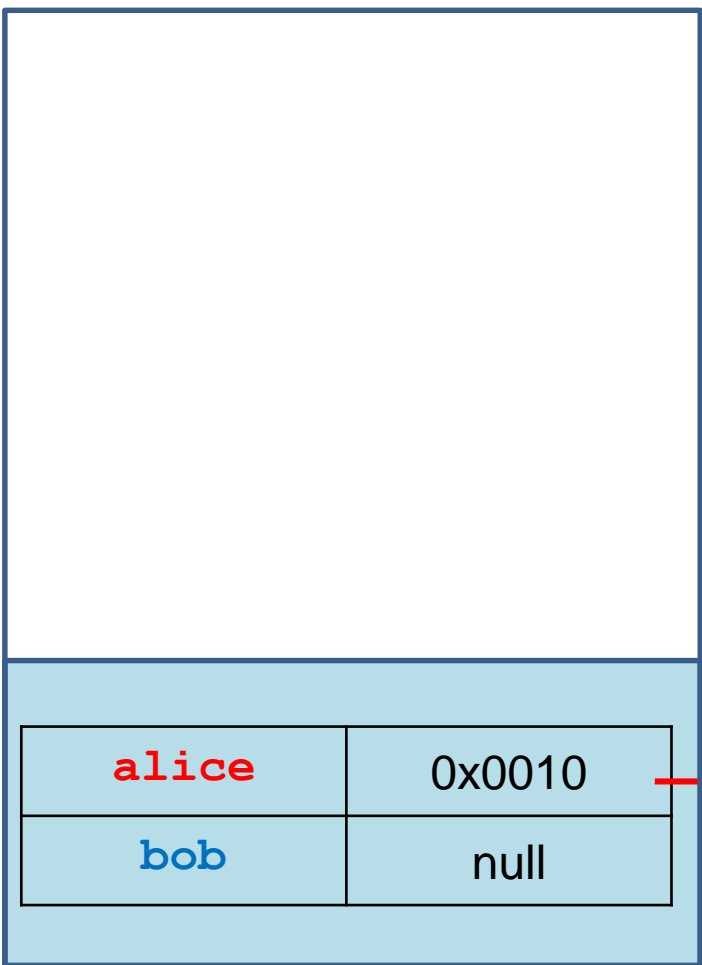
Η από-αναφοροποίηση (dereferencing) γίνεται ουσιαστικά μέσω του "." το οποίο μπορούμε να σκεφτούμε σαν να ακολουθεί τα βελάκια στο παρακάτω σχήμα



Dereferencing

Ένα αντικείμενο αποθηκεύεται σαν μια μεταβλητή η οποία κρατάει μια αναφορά η οποία «δείχνει» στην θέση μνήμης που αποθηκεύει το αντικείμενο.

Η από-αναφοροποίηση (dereferencing) γίνεται ουσιαστικά μέσω του "." το οποίο μπορούμε να σκεφτούμε σαν να ακολουθεί τα βελάκια στο παρακάτω σχήμα

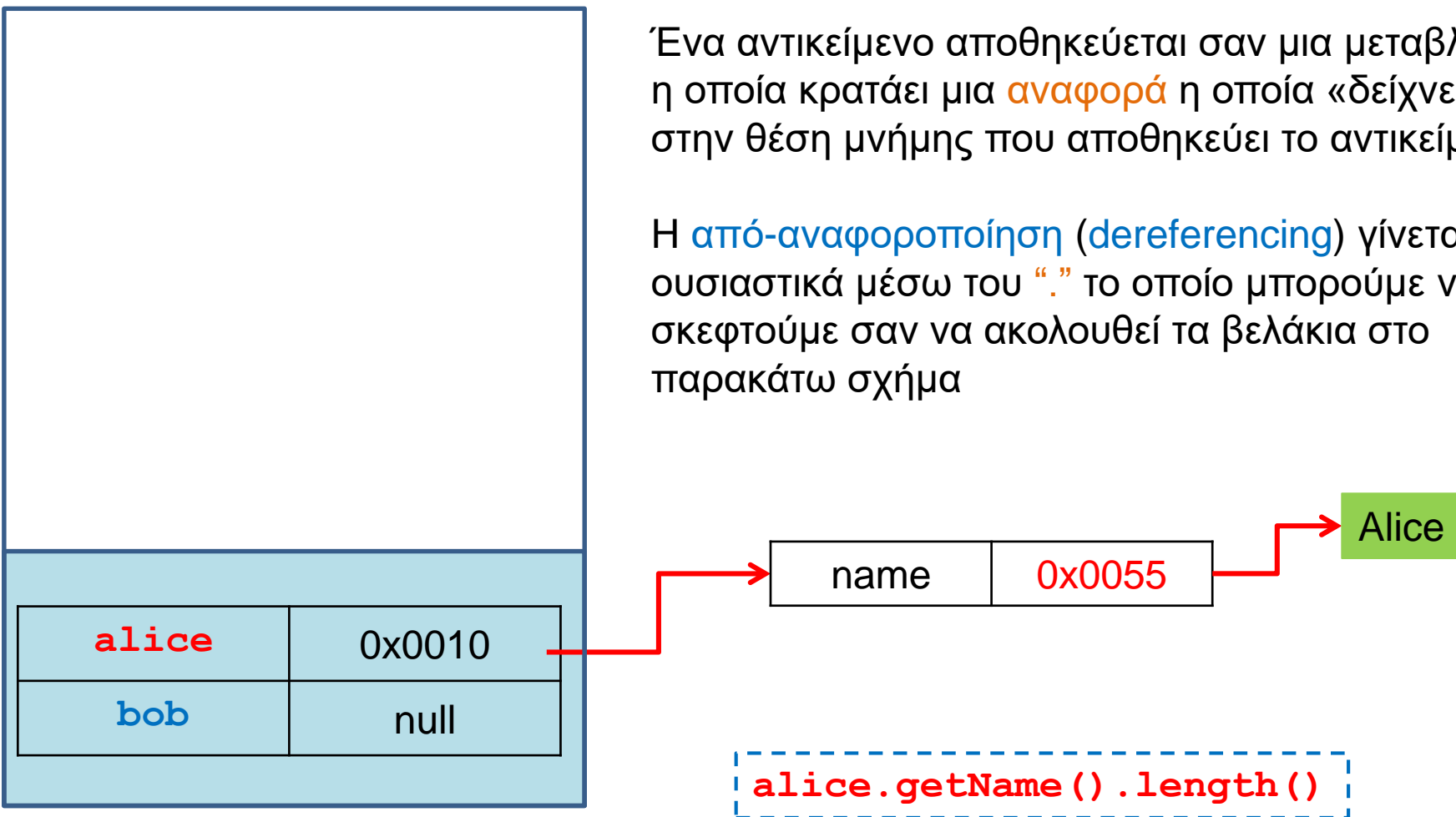


```
alice.getName().length()
```

Dereferencing

Ένα αντικείμενο αποθηκεύεται σαν μια μεταβλητή η οποία κρατάει μια αναφορά η οποία «δείχνει» στην θέση μνήμης που αποθηκεύει το αντικείμενο.

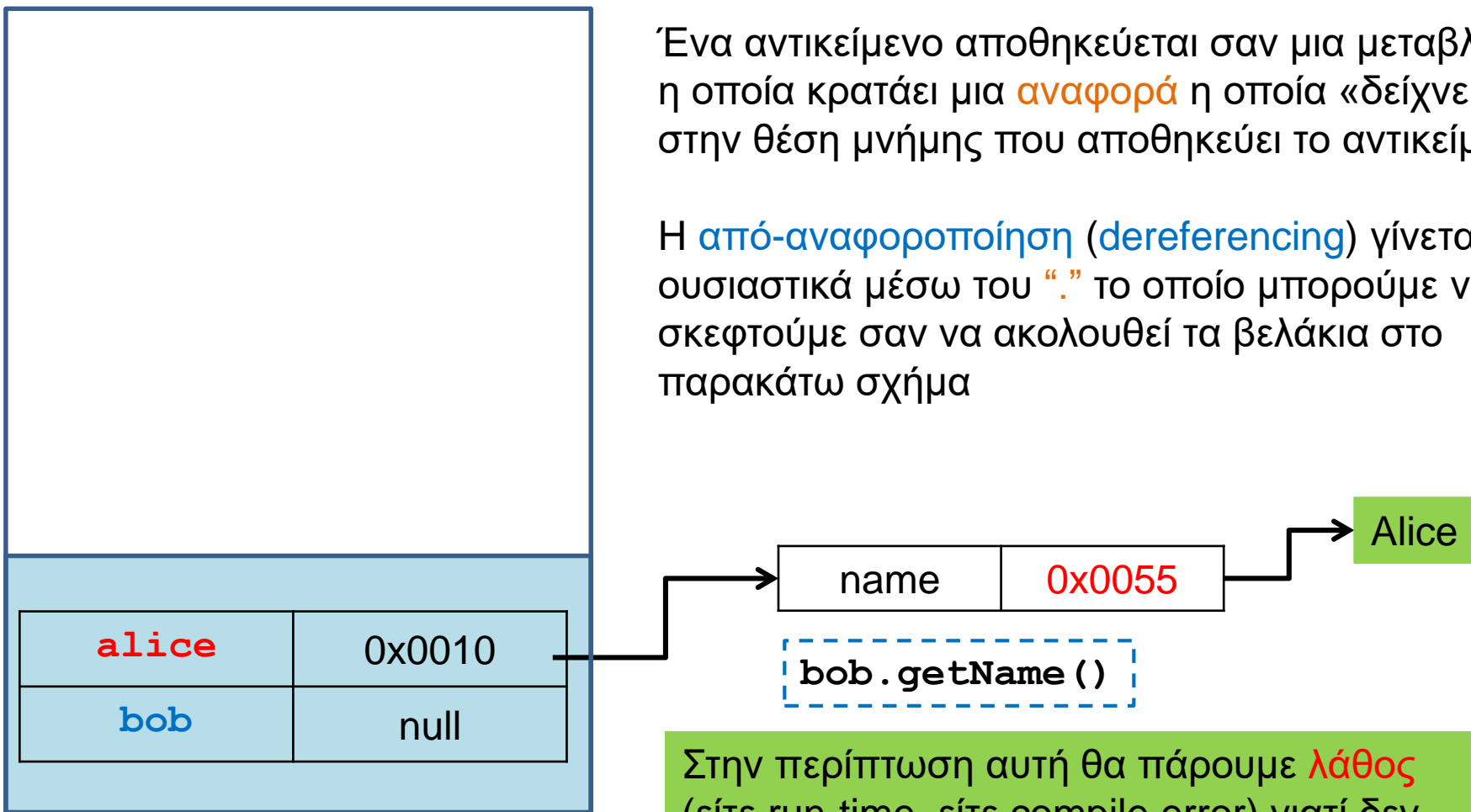
Η από-αναφοροποίηση (dereferencing) γίνεται ουσιαστικά μέσω του "." το οποίο μπορούμε να σκεφτούμε σαν να ακολουθεί τα βελάκια στο παρακάτω σχήμα



Dereferencing

Ένα αντικείμενο αποθηκεύεται σαν μια μεταβλητή η οποία κρατάει μια αναφορά η οποία «δείχνει» στην θέση μνήμης που αποθηκεύει το αντικείμενο.

Η από-αναφοροποίηση (dereferencing) γίνεται ουσιαστικά μέσω του "." το οποίο μπορούμε να σκεφτούμε σαν να ακολουθεί τα βελάκια στο παρακάτω σχήμα



Στην περίπτωση αυτή θα πάρουμε **λάθος** (είτε run-time, είτε compile error) γιατί δεν υπάρχει διεύθυνση να ακολουθήσουμε

```
class Person
```

```
{  
    private String name;  
  
    public Person(String name){  
        this.name = name;  
    }  
  
    public String getName(){  
        return name;  
    }  
}
```

```
class Car
```

```
{  
    private int position = 0;  
    private Person driver;  
  
    public Car(int position, Person driver){  
        this.position = position;  
        this.driver = driver;  
    }  
  
    public Person getDriver(){  
        return driver;  
    }  
}
```

```
class MovingCarDriver1
```

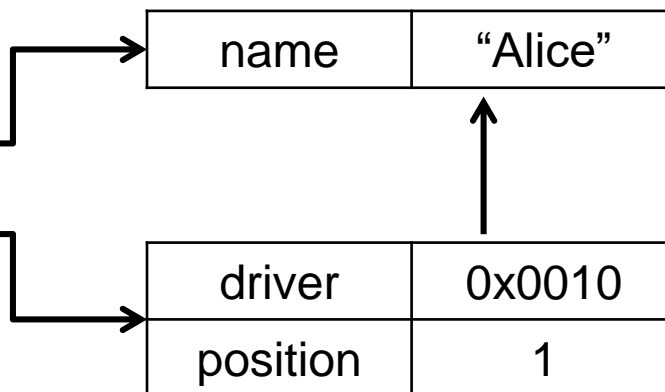
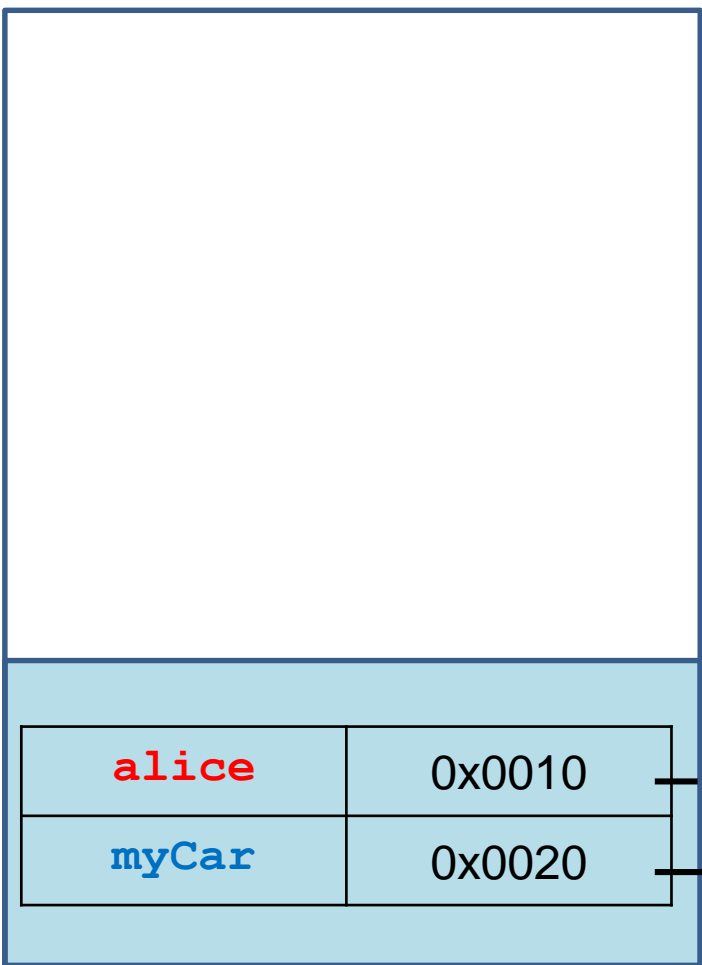
```
{  
    public static void main(String args[])  
    {  
        Person alice = new Person("Alice");  
        Car myCar = new Car(1, alice);  
        System.out.println(myCar.getDriver().getName());  
    }  
}
```

Dereferencing

Στην περίπτωση αυτή έχουμε ένα αντικείμενο μέσα σε ένα άλλο αντικείμενο.
Η μέθοδος `getDriver()` επιστρέφει αντικείμενο `Person`

Έχουμε αλυσιδωτή πρόσβαση σε αναφορές

```
myCar.getDriver().getName()
```

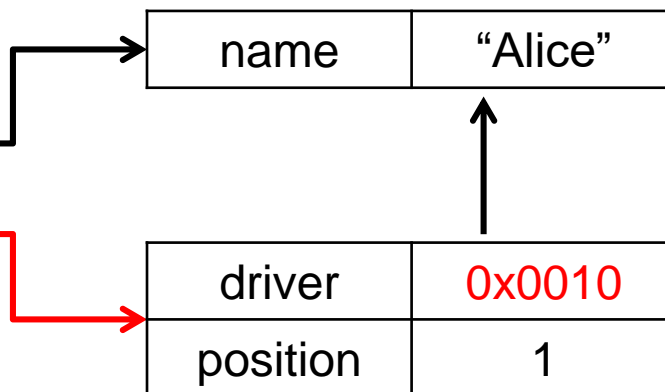
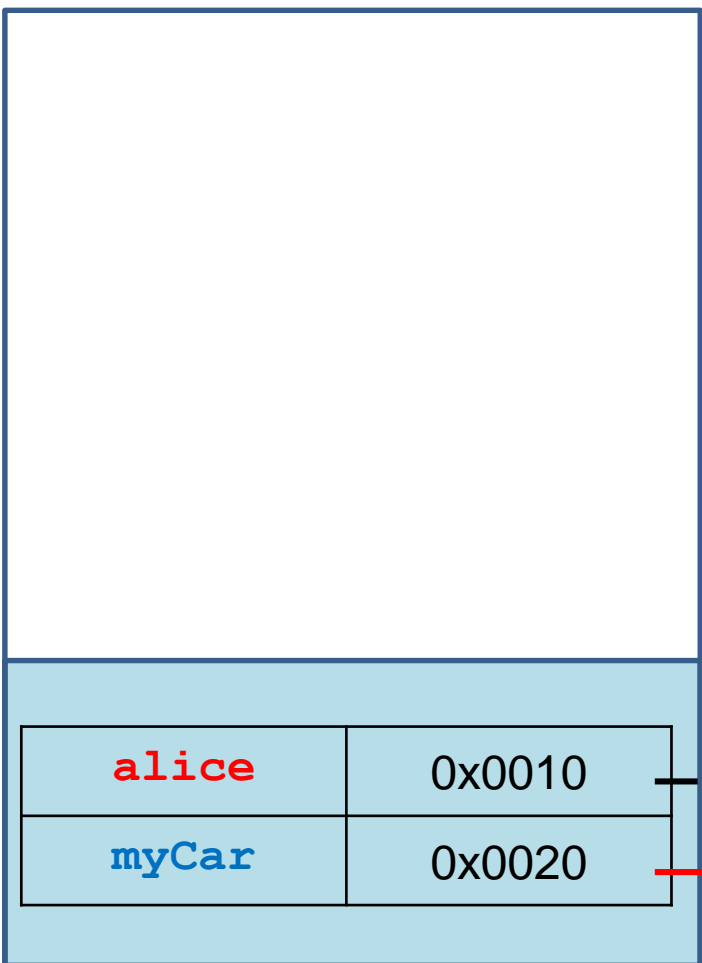


Dereferencing

Στην περίπτωση αυτή έχουμε ένα αντικείμενο μέσα σε ένα άλλο αντικείμενο.
Η μέθοδος `getDriver()` επιστρέφει αντικείμενο `Person`

Έχουμε αλυσιδωτή πρόσβαση σε αναφορές

```
myCar.getDriver().getName()
```

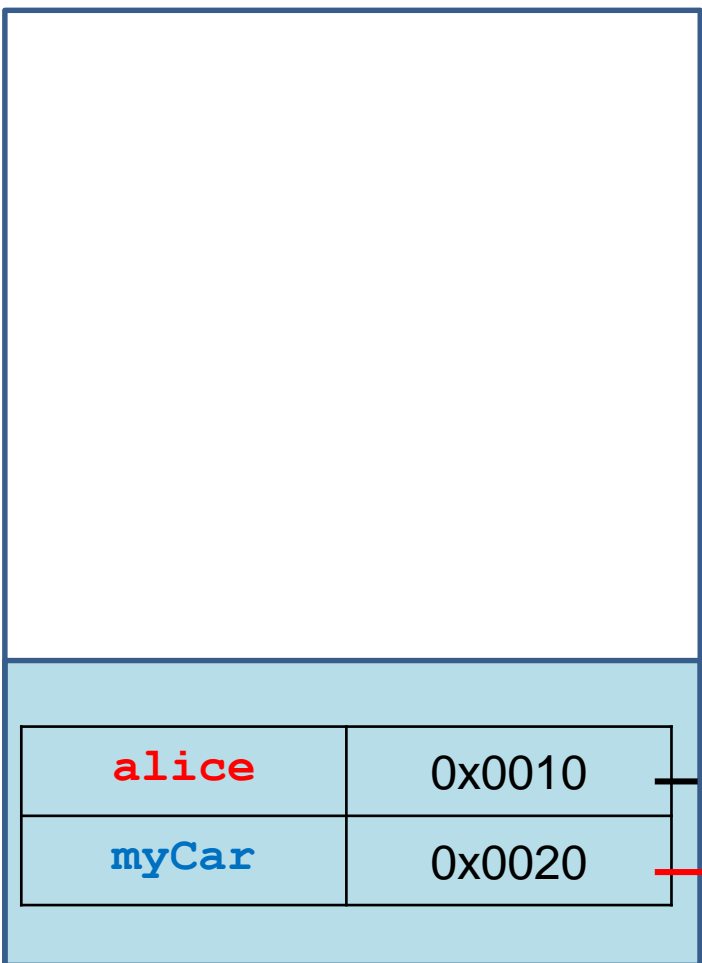


Dereferencing

Στην περίπτωση αυτή έχουμε ένα αντικείμενο μέσα σε ένα άλλο αντικείμενο.
Η μέθοδος `getDriver()` επιστρέφει αντικείμενο `Person`

Έχουμε αλυσιδωτή πρόσβαση σε αναφορές

```
myCar.getDriver().getName()
```



<code>name</code>	<code>"Alice"</code>
-------------------	----------------------

<code>driver</code>	<code>0x0010</code>
<code>position</code>	<code>1</code>

```
class Person
```

```
{  
    private String name;  
  
    public Person(String name){  
        this.name = name;  
    }  
  
    public String getName () {  
        return name;  
    }  
}
```

```
class Car
```

```
{  
    private int position = 0;  
    private Person driver;  
  
    public Car(int position, String name) {  
        this.position = position;  
        this.driver = new Person(name);  
    }  
  
    public String getDriverName () {  
        return driver.getName ();  
    }  
}
```

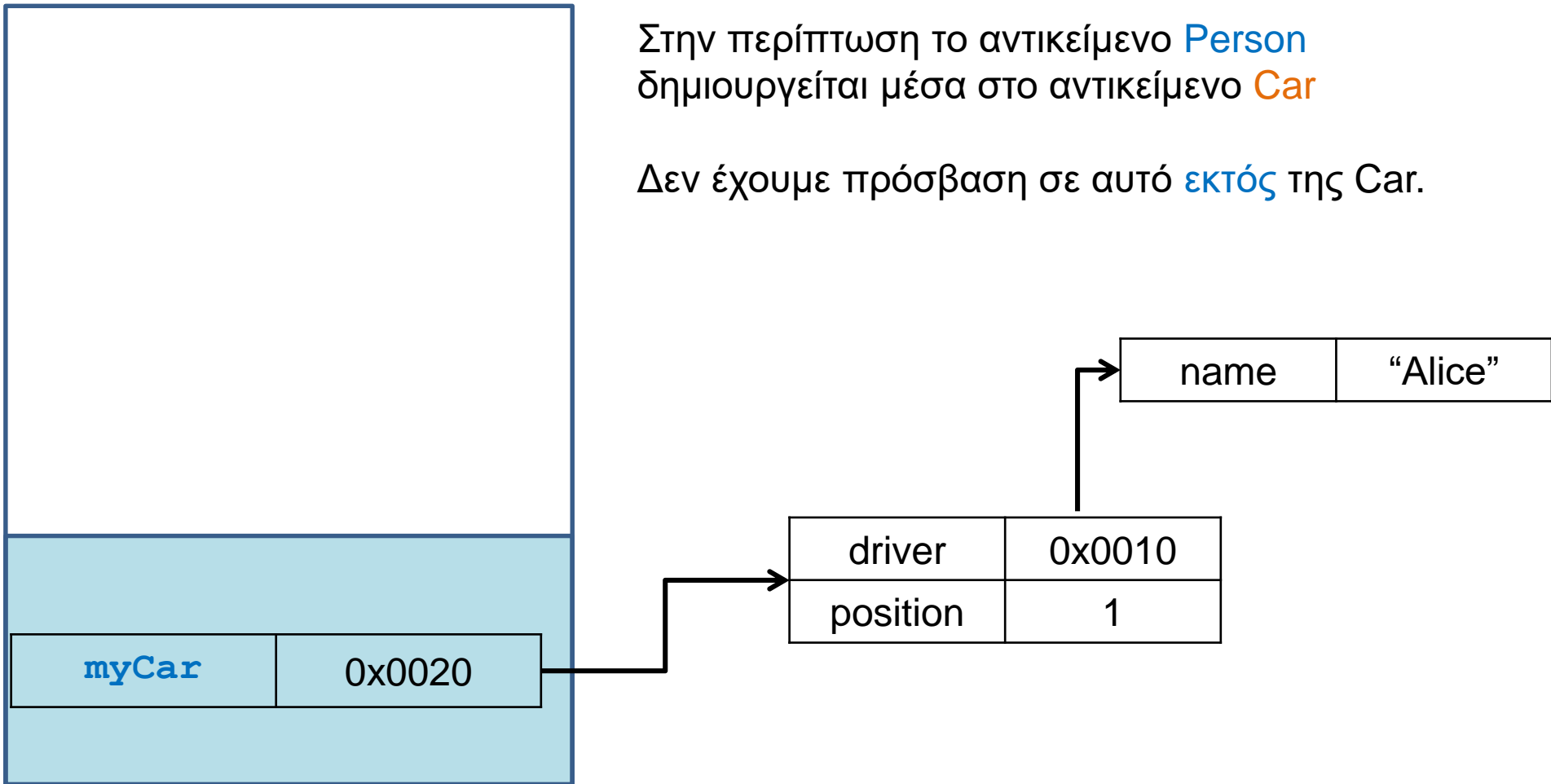
```
class MovingCarDriver2
```

```
{  
    public static void main(String args[])  
    {  
        Car myCar = new Car(1, "Alice");  
        System.out.println(myCar.getDriverName());  
    }  
}
```

Αντικείμενα μέσα σε αντικείμενα

Στην περίπτωση το αντικείμενο **Person** δημιουργείται μέσα στο αντικείμενο **Car**

Δεν έχουμε πρόσβαση σε αυτό **εκτός** της Car.



Σχέσεις μεταξύ κλάσεων

- Στο παράδειγμα μας έχουμε δύο διαφορετικές κλάσεις (**Person**, **Driver**) οι οποίες συσχετίζονται μεταξύ τους με διαφορετικούς τρόπους.
- Μπορεί να υπάρχουν πολλές διαφορετικές σχέσεις μεταξύ κλάσεων.
 - Στην περίπτωση μας, η μία κλάση ορίζεται χρησιμοποιώντας αντικείμενα της άλλης
- Αυτού του είδους τη σχέση την λέμε σχέση **σύνθεσης**
 - Μερικές φορές την ξεχωρίζουμε σε σχέση **σύνθεσης** (composition) και **συνάθροισης** (aggregation).

Σχέσεις κλάσεων

- Όταν έχουμε **κλάσεις** που **έχουν αντικείμενα άλλων κλάσεων** ένα θέμα που προκύπτει είναι πότε και πού θα γίνεται η **δημιουργία των αντικειμένων** και πότε η καταστροφή τους
 - Πιο σημαντικό σε γλώσσες που δεν έχουν garbage collector.
- Π.χ., τα αντικείμενα τύπου **Person** στο παράδειγμα **MovingCarDriver2** **δημιουργούνται μέσα** στην κλάση **Car**, και καταστρέφονται μέσα στην **Car**, ή αν το αντικείμενο **Car** καταστραφεί.
- Τα αντικείμενα τύπου **Person** που χρησιμοποιούνται στην **MovingCarDriver1** **δημιουργούνται εκτός της κλάσης** και μπορεί να υπάρχουν αφού καταστραφεί η κλάση.
- Συχνά οι σχέσεις του δεύτερου τύπου λέγονται σχέσεις **συνάθροισης**, ενώ σχέσεις του πρώτου τύπου, **σύνθεσης**.

ΑΝΤΙΚΕΙΜΕΝΑ ΩΣ ΕΠΙΣΤΡΕΦΟΜΕΝΕΣ ΤΙΜΕΣ

Επιστροφή αντικειμένων

- Ένα **αντικείμενο** που δημιουργούμε **μέσα σε μία μέθοδο** μπορούμε να το διατηρήσουμε και μετά το τέλος της μεθόδου αν **κρατήσουμε μια αναφορά** σε αυτό.
- Ένας τρόπος να γίνει αυτό είναι αν η μέθοδος **επιστρέφει** το αντικείμενο (δηλαδή την **αναφορά** σε αυτό) που δημιουργήσαμε


```
class Date
```

```
{
```

```
    private int day = 1;
```

```
    private int month = 1;
```

```
    private int year = 2015;
```

```
    private String[] monthStrings =
```

```
        {"Jan", "Feb", "Mar", "Apr", "May", "Jun",  
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
```

```
    public Date(int day, int month, int year) {
```

```
        this.day = day;
```

```
        this.month = month;
```

```
        this.year = year;
```

```
    }
```

```
    public String toString() {
```

```
        return day + " " + monthNames[month-1] + " " + year;
```

```
    }
```

```
}
```

Η κλάση Date

Θέλω η κλάση να μπορεί να μου επιστρέφει μια νέα ημερομηνία αλλά ένα χρόνο μετά. Πως μπορώ να το κάνω?

```
class Date
```

```
{
```

```
    private int day = 1;
```

```
    private int month = 1;
```

```
    private int year = 2014;
```

```
    private String[] monthStrings =
```

```
        {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
```

```
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
```

```
    public Date(int day, int month, int year){
```

```
        this.day = day; this.month = month; this.year = year;
```

```
    }
```

```
    public String toString(){
```

```
        return day + " " + monthNames[month-1] + " " + year;
```

```
    }
```

```
    public Date nextYear(){
```

```
        Date nextYearDate = new Date(day, month, year+1);
```

```
        return nextYearDate;
```

```
    }
```

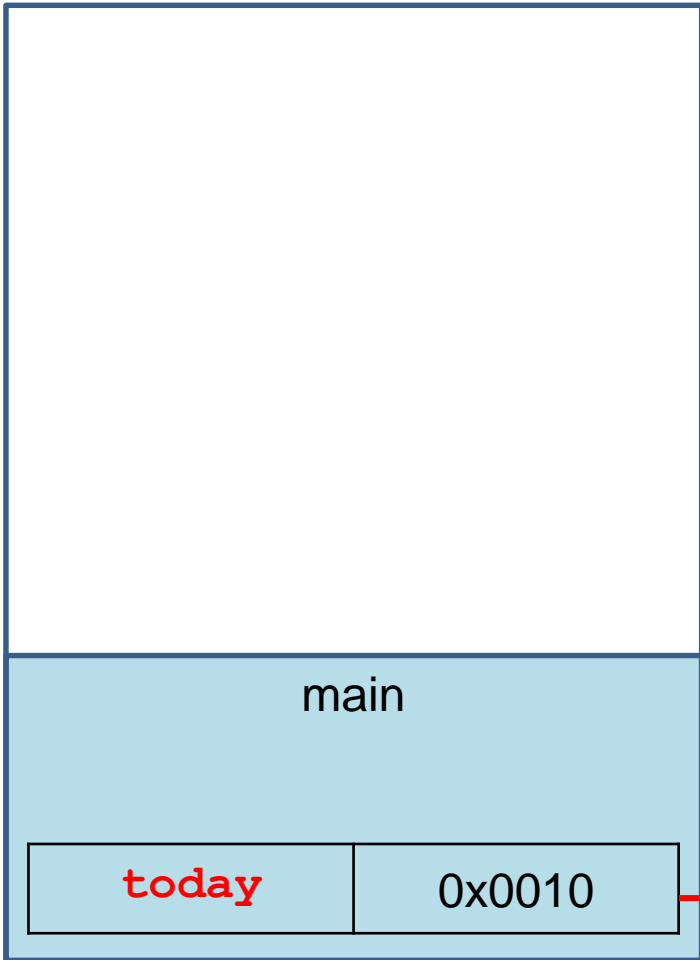
```
}
```

Η κλάση Date

Η κλάση nextYear() επιστρέφει ένα νέο αντικείμενο Date με την ημερομηνία ένα χρόνο μετά.

```
class DateExample
{
    public static void main(String args[]) {
        Date today = new Date(16,4,2018);
        System.out.println(today);
        Date todayNextYear = today.nextYear();
        System.out.println(todayNextYear);
    }
}
```

Εξέλιξη του προγράμματος



```
class DateExample
{
    public static void main(String args[]) {
        Date today = new Date(16,4,2018);
        System.out.println(today);
        Date todayNextYear = today.nextYear();
        System.out.println(todayNextYear);
    }
}
```

day	16
month	4
year	2018
monthStrings	0x0050



Εξέλιξη του προγράμματος

Η πρόσβαση στα πεδία day, month, year είναι μέσω της μεταβλητής this

```
public Date nextYear(){  
    Date nextYearDate =  
        new Date(day, month, year+1);  
    return nextYearDate;  
}
```

today.nextYear()

nextYearDate	0x0200
this	0x0010

main

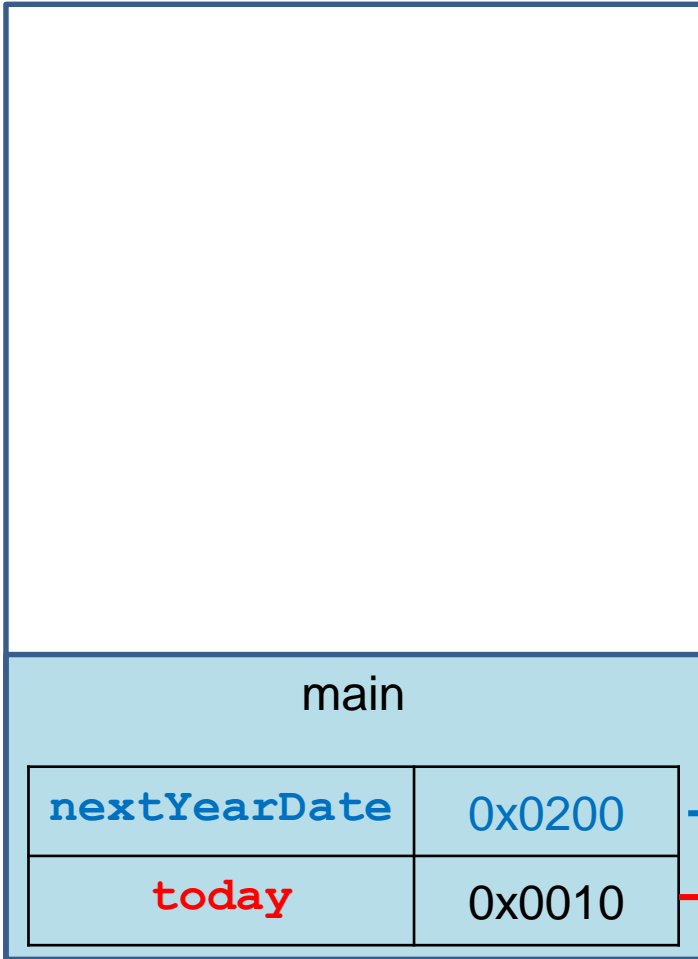
today	0x0010
-------	--------

day	16
month	4
year	2019
monthStrings	0x0250

day	16
month	4
year	2018
monthStrings	0x0050

Εξέλιξη του προγράμματος

```
class DateExample
{
    public static void main(String args[]) {
        Date today = new Date(16,4,2018);
        System.out.println(today);
        Date todayNextYear = today.nextYear();
        System.out.println(todayNextYear);
    }
}
```



day	16
month	4
year	2019
monthStrings	0x0250

Blue arrows point from the `monthStrings` field to a small table with three empty cells.

day	16
month	4
year	2018
monthStrings	0x0050

Blue arrows point from the `monthStrings` field to a small table with three empty cells.

```
class Date
```

```
{
```

```
    private int day = 1;
```

```
    private int month = 1;
```

```
    private int year = 2014;
```

```
    private String[] monthStrings =
```

```
        {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
```

```
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
```

```
    public Date(int day, int month, int year){
```

```
        this.day = day; this.month = month; this.year = year;
```

```
    }
```

```
    public String toString(){
```

```
        return day + " " + monthNames[month-1] + " " + year;
```

```
    }
```

```
    public Date nextYear(){
```

```
        return new Date(day, month, year+1);
```

```
    }
```

```
}
```

Η κλάση Date

Τι γίνεται αν η ημερομηνία είναι 29/2?

Μπορούμε να επιστρέψουμε το αντικείμενο που δημιουργούμε κατευθείαν ως επιστρεφόμενη τιμή (παρομοίως και ως όρισμα σε μέθοδο)

class Date

```
{
    private int day = 1;
    private int month = 1;
    private int year = 2014;
    private String[] monthStrings =
        {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    public Date(int day, int month, int year){
        this.day = day; this.month = month; this.year = year;
    }

    public String toString(){
        return day + " " + monthNames[month-1] + " " + year;
    }

    public Date nextYear(){
        if (day == 29 && month == 2){
            return null;
        }
        return new Date(day, month, year+1);
    }
}
```

Η κλάση Date

Τι γίνεται αν η ημερομηνία είναι 29/2?

Η τιμή null: Μία κενή αναφορά. Η τιμή μπορεί να χρησιμοποιηθεί σαν μια default τιμή, ή σαν ένδειξη λάθους (στην περίπτωση αυτή ότι δεν μπορούμε να δημιουργήσουμε το αντικείμενο)


```
class DateExample
{
    public static void main(String args[]) {
        Date today = new Date(3,4,2014);
        System.out.println(today);
        Date todayNextYear = today.nextYear();
        if( todayNextYear != null) {
            System.out.println(todayNextYear);
        }
    }
}
```

Προσοχή: Η χρήση του null για έλεγχο λάθους σημαίνει ότι όποτε χρησιμοποιούμε την μέθοδο θα πρέπει να προσέχουμε αν η επιστρεφόμενη τιμή είναι null. Δεν είναι καλή λύση, και αργότερα θα μάθουμε για εξαιρέσεις για να χειριζόμαστε τέτοια προβλήματα.

ΠΙΝΑΚΕΣ ΑΠΟ ΑΝΤΙΚΕΙΜΕΝΑ

Πίνακες από αντικείμενα

- Όπως ορίζουμε πίνακες από πρωταρχικούς τύπους μπορούμε να ορίσουμε και **πίνακες από αντικείμενα**
 - `Person[] array = new Person[3];`
 - Ορίζει ένα πίνακα με τρία αντικείμενα τύπου Person
 - Ουσιαστικά ένα πίνακα με **αναφορές**.
- Όταν ορίζουμε ένα πίνακα από αντικείμενα πρέπει να είμαστε προσεκτικοί να δεσμεύουμε σωστά τη μνήμη.

Παράδειγμα

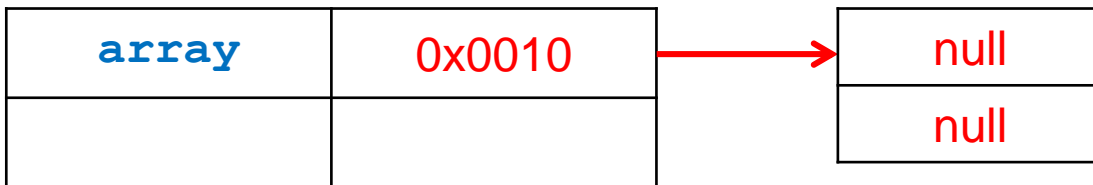
```
Person[] array;
```

<code>array</code>	null

- Η εντολή αυτή θα δημιουργήσει μια μεταβλητή με το όνομα `array` η οποία κάποια στιγμή θα δείχνει σε ένα πίνακα με `Person`. Για την ώρα είναι `null`.

Παράδειγμα

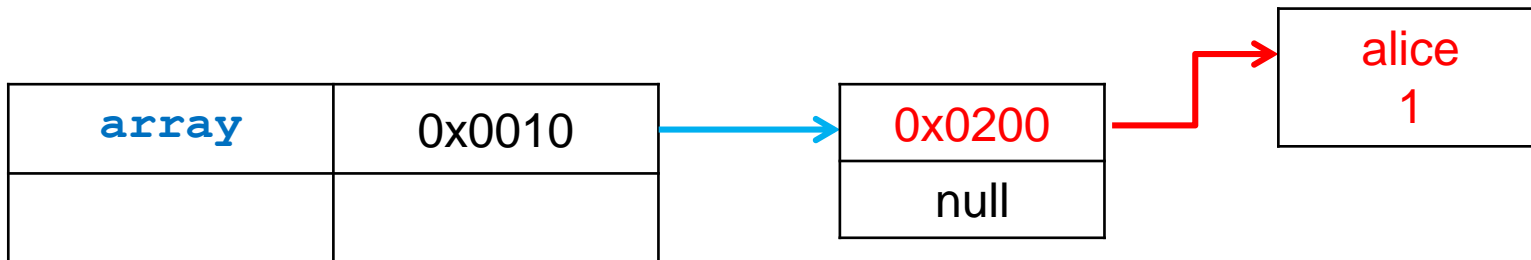
```
Person[] array;  
array = new Person[2];
```



- Η εντολή `new` θα δεσμεύσει δύο θέσεις μνήμης στο `heap` για να κρατήσουν δύο αναφορές τύπου `Person`. Εφόσον δεν έχουμε δημιουργήσει τις μεταβλητές ακόμη, αυτές θα είναι `null`.

Παράδειγμα

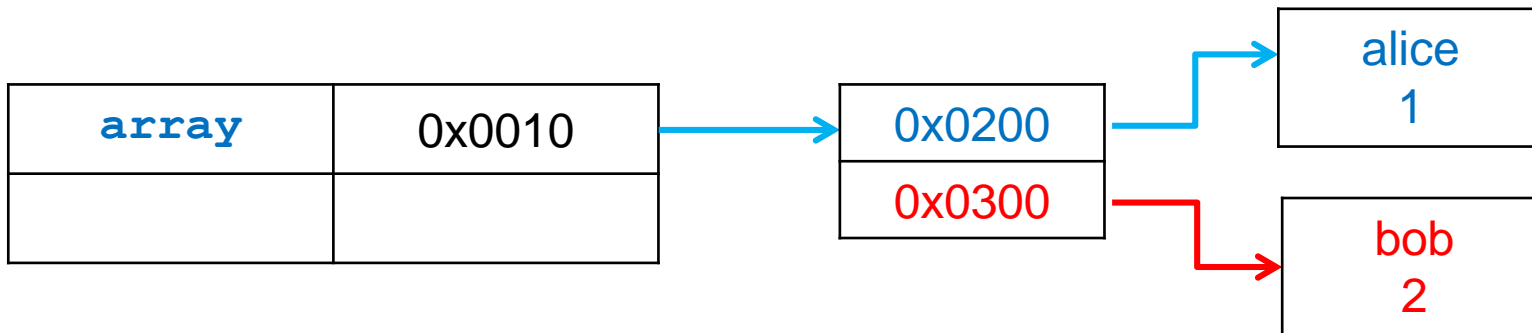
```
Person[] array;  
array = new Person[2];  
array[0] = new Person("alice", 1);
```



- Η νέα εντολή `new` θα δεσμεύσει χώρο για ένα `Person`. Δημιουργείται το αντικείμενο και η αναφορά αποθηκεύεται στην πρώτη θέση του πίνακα `array`.

Παράδειγμα

```
Person[] array;  
array = new Person[2];  
array[0] = new Person("alice", 1);  
array[1] = new Person("bob", 1);
```



- Η νέα εντολή `new` θα δεσμεύσει χώρο για άλλο ένα `Person`. Δημιουργείται το αντικείμενο και η αναφορά αποθηκεύεται στην δεύτερη θέση του πίνακα `array`.

Πίνακες από πίνακες

- Οι δισδιάστατοι πίνακες είναι ουσιαστικά πίνακες από αντικείμενα, όπου τα αντικείμενα είναι πάλι πίνακες
- Π.χ., έτσι δεσμεύουμε πίνακα ακεραίων 5×5

```
int[][] array;  
array = new int[5][];  
for (int i=0; i<5; i++) {  
    array[i] = new int[5];  
}
```

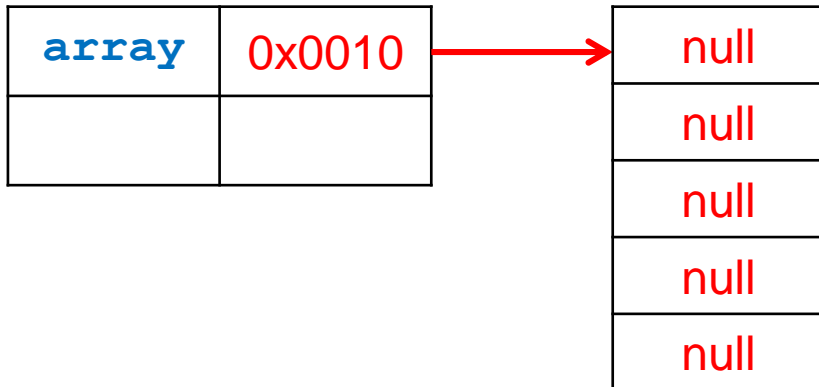

Παράδειγμα

```
int[][] array;
```

array	null

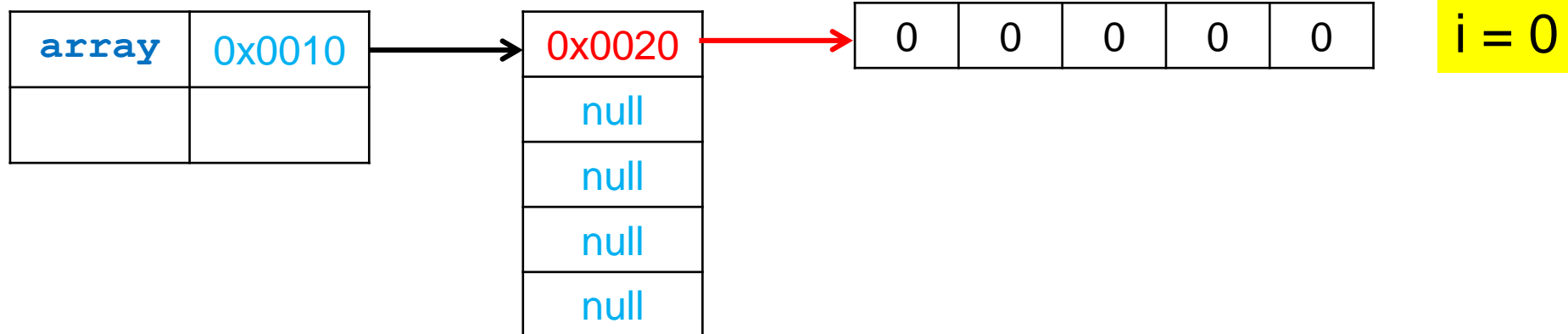
Παράδειγμα

```
int[][] array;  
array = new int[5][];
```



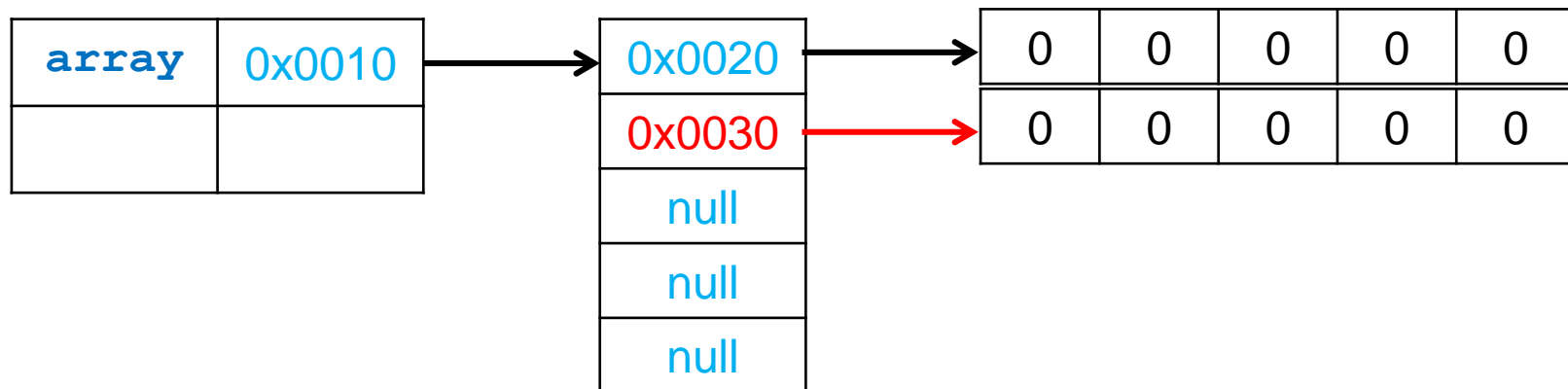
Παράδειγμα

```
int[][] array;  
array = new int[5][];  
for (int i=0; i<5; i++){  
    array[i] = new int[5];  
}
```



Παράδειγμα

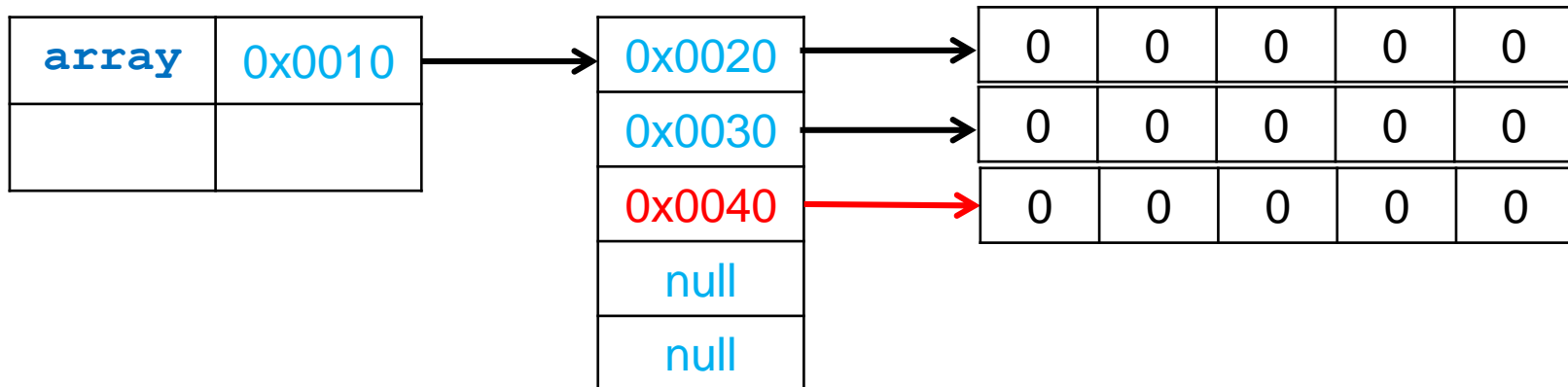
```
int[][] array;  
array = new int[5][];  
for (int i=0; i<5; i++){  
    array[i] = new int[5];  
}
```



i = 1

Παράδειγμα

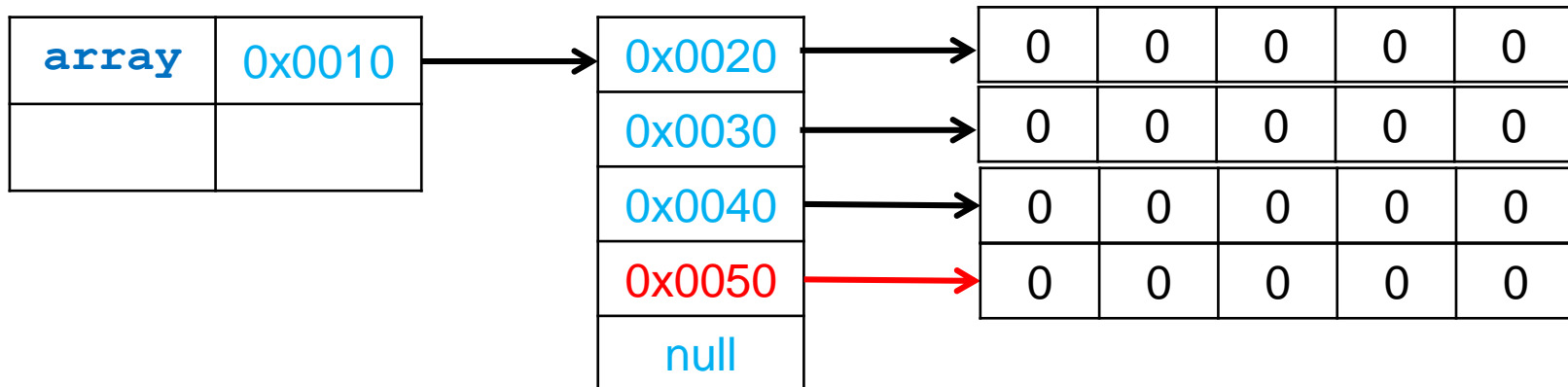
```
int[][] array;  
array = new int[5][];  
for (int i=0; i<5; i++){  
    array[i] = new int[5];  
}
```



$i = 2$

Παράδειγμα

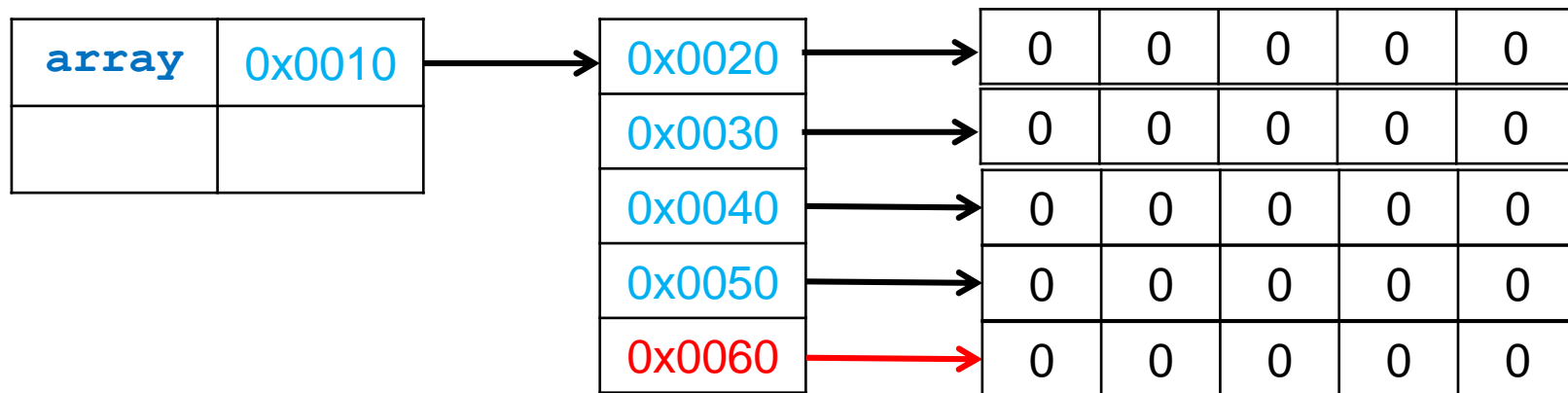
```
int[][] array;  
array = new int[5][];  
for (int i=0; i<5; i++){  
    array[i] = new int[5];  
}
```



i = 3

Παράδειγμα

```
int[][] array;  
array = new int[5][];  
for (int i=0; i<5; i++){  
    array[i] = new int[5];  
}
```



$i = 4$

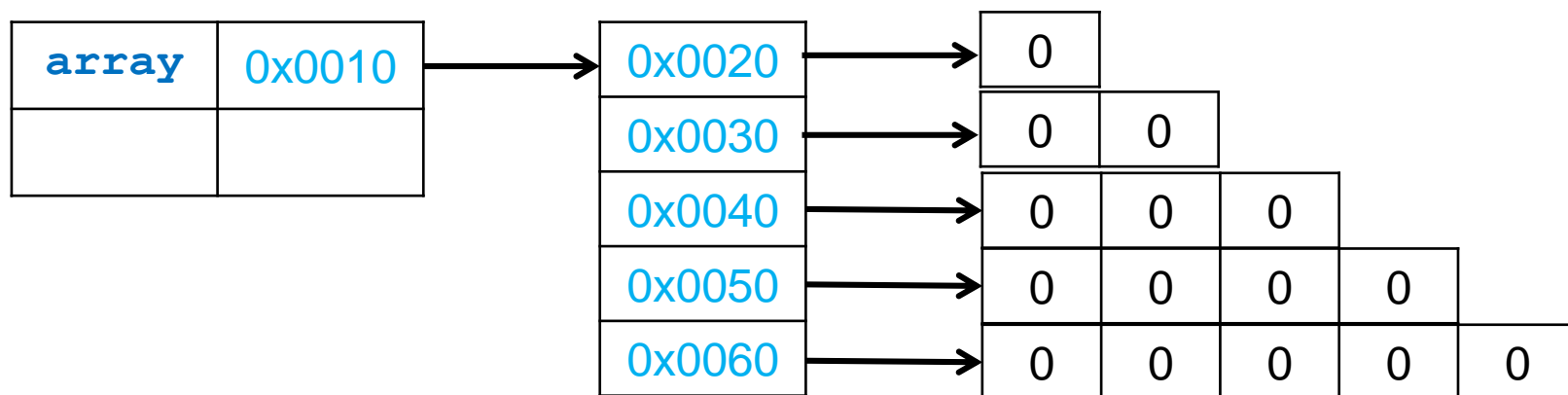
Πίνακες από πίνακες

- Μπορεί ο δισδιάστατος μας πίνακας να είναι ασύμμετρος.
- Π.χ., έτσι ορίζουμε ένα διαγώνιο πίνακα.

```
int[][] array;  
array = new int[5][];  
for (int i=0; i<5; i++) {  
    array[i] = new int[i+1];  
}
```


Παράδειγμα

```
int[][] array;  
array = new int[5][];  
for (int i=0; i<5; i++){  
    array[i] = new int[i+1];  
}
```



ΒΑΘΕΙΑ ΚΑΙ ΡΗΧΑ ΑΝΤΙΓΡΑΦΑ COPY CONSTRUCTOR

```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber) {
        name = initName;
        number = initNumber;
    }

    public void set(String newName, int newNumber) {
        name = newName;
        number = newNumber;
    }

    public String toString() {
        return (name + " " + number);
    }

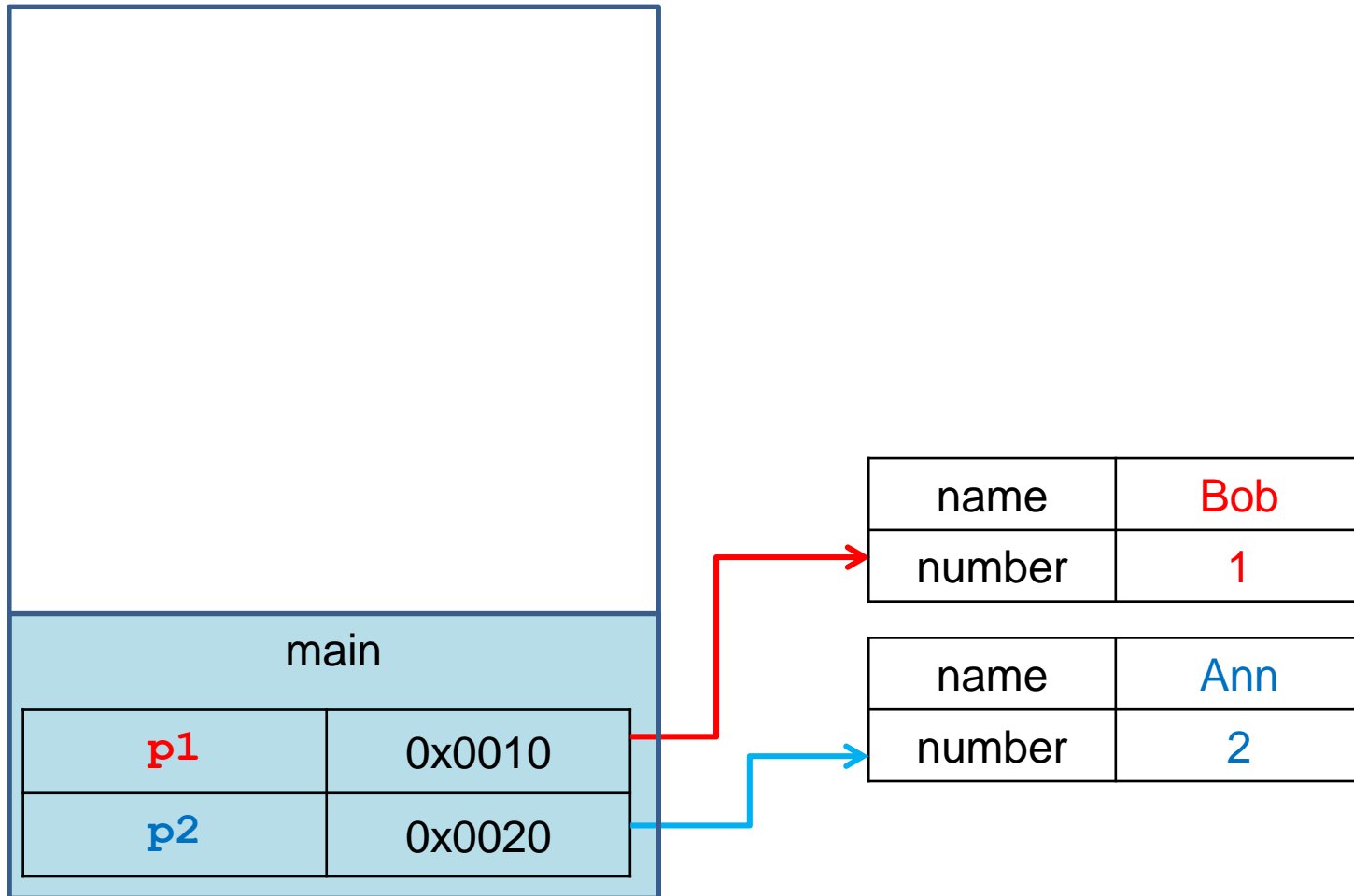
    public Person copier() {
        Person newPerson = new Person(this.name, this.number);
        return newPerson;
    }
}
```

Παράδειγμα

```
public class ClassParameterDemo2
{
    public static void main(String[] args)
    {
        Person p1 = new Person("Bob", 1);
        Person p2 = new Person("Ann", 2);
        p1 = p2.copier();
        System.out.println(p1);
    }
}
```

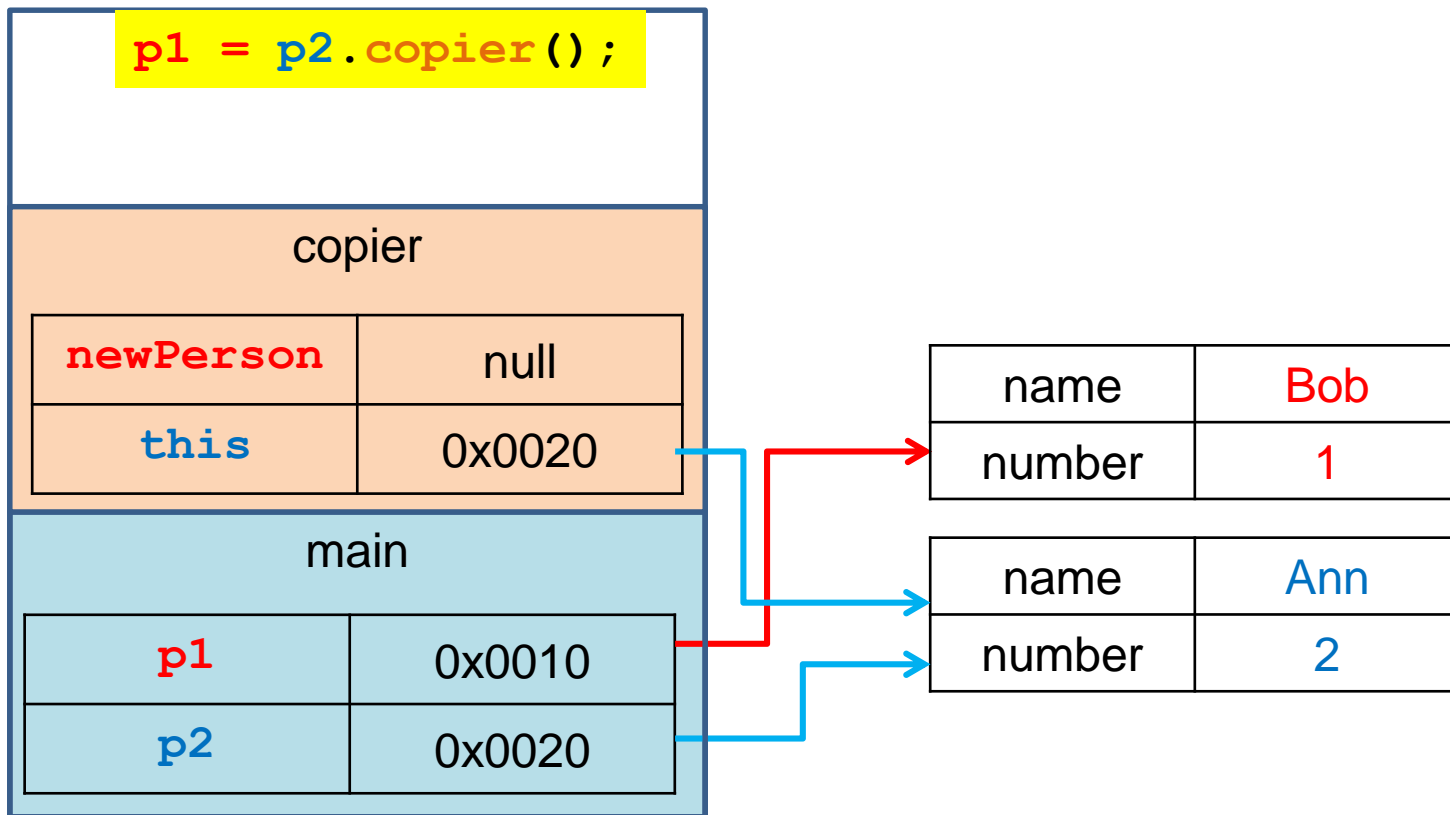
Τι θα τυπώσει?

Εξέλιξη του προγράμματος



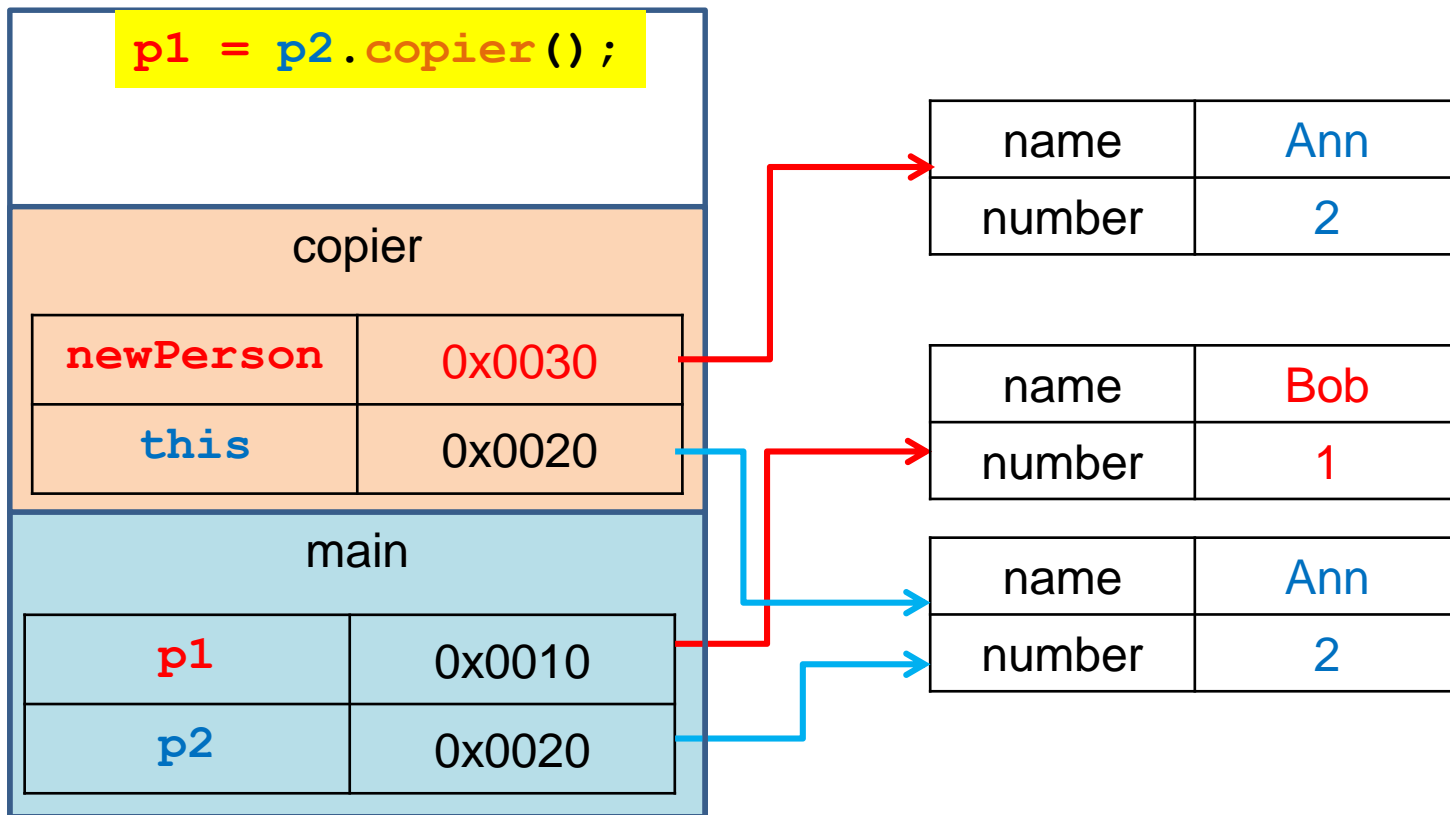
Εξέλιξη του προγράμματος

```
public Person copier() {  
    Person newPerson = new Person(this.name, this.number);  
    return newPerson;  
}
```



Εξέλιξη του προγράμματος

```
public Person copier() {  
    Person newPerson = new Person(this.name, this.number);  
    return newPerson;  
}
```



Εξέλιξη του προγράμματος

```
public Person copier() {  
    Person newPerson = new Person(this.name, this.number);  
    return newPerson;  
}
```

```
p1 = p2.copier();
```

main

p1

0x0030

p2

0x0020

name

Ann

number

2

name

Bob

number

1

name

Ann

number

2

Η main τυπώνει "Ann 2"

Εξέλιξη του προγράμματος

```
public Person copier() {  
    Person newPerson = new Person(this.name, this.number);  
    return newPerson;  
}
```

```
p1 = p2.copier();
```

main

p1

0x0030

p2

0x0020

name

Ann

number

2

~~name~~

~~Bob~~

~~number~~

~~1~~

name

Ann

number

2

Το προηγούμενο αντικείμενο αποδεσμεύεται

Δημιουργία αντιγράφων

- Η μέθοδος **copier** όπως την ορίσαμε πριν δημιουργεί ένα **καινούριο αντικείμενο** που είναι **αντίγραφο** αυτού που έκανε την κλήση.
- Στην περίπτωση μας το αντικείμενο έχει μόνο πεδία που είναι **πρωταρχικού τύπου** ή **μη μεταλλάξιμα αντικείμενα**. Γενικά ένα αντικείμενο μπορεί να έχει ως πεδία άλλα **αντικείμενα** (δηλαδή αναφορές).
- Στην περίπτωση αυτή η **δημιουργία αντιγράφου** θα πρέπει να γίνεται με πολύ **προσοχή!**

```
class Car
{
    private int dim;
    private int[] position;

    public Car(int d){
        dim = d; position = new int[d];
    }

    public void move(){
        for (int i=0; i < dim; i++){position[i] ++;}
    }
}
```

Το Car κινείται σε 1 ή 2 διαστάσεις
Χρειαζόμαστε ένα πίνακα για την θέση του

```
public Car copy(){
    Car newCar = new Car(this.dim);
    newCar.position = this.position;
    return newCar;
}
```

Η copy δημιουργεί και επιστρέφει ένα νέο Car

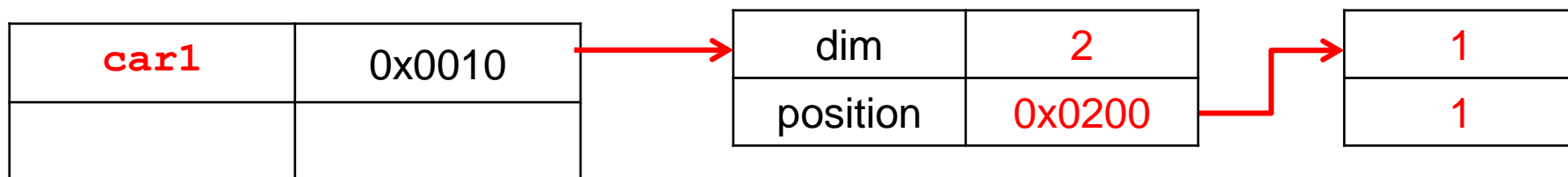
```
public String toString(){
    String output = "";
    for (int i=0; i < dim; i++){output = output + position[i] + " ";}
    return output;
}
```

```
public static void main(String args[]){
    Car car1 = new Car(2);
    car1.move();
    Car car2 = car1.copy();
    car2.move();
    System.out.println(car1);
}
```

Τι θα τυπώσει η main?

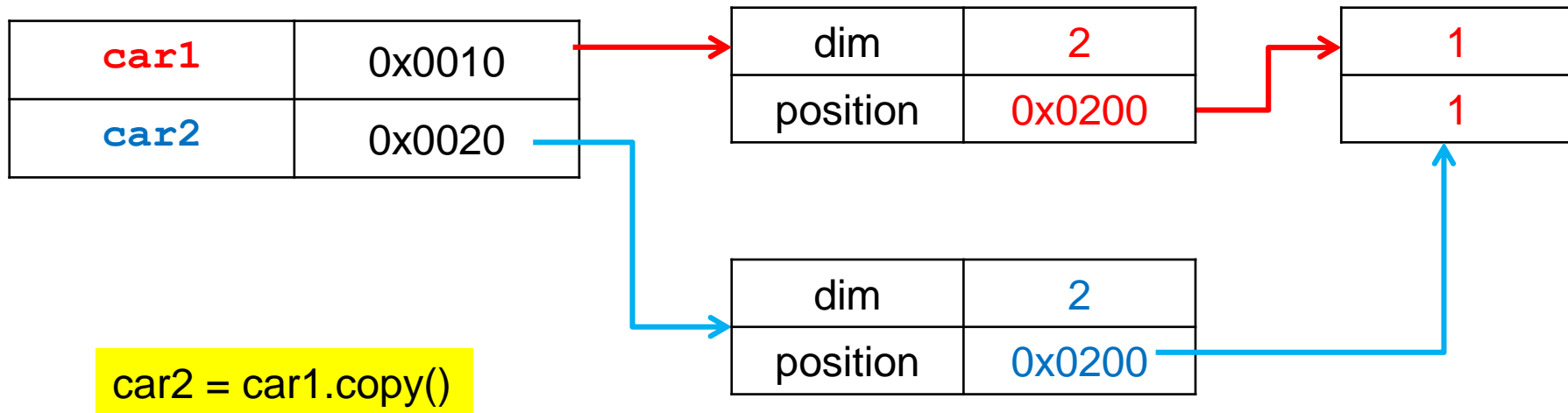
Ρηχά Αντίγραφα

- Η copy όπως την έχουμε ορίσει δημιουργεί ένα **ρηχό αντίγραφο** του αντικειμένου
 - Αντιγράφει τις **αναφορές** στα αντικείμενα και όχι τα **περιεχόμενα** των αντικειμένων



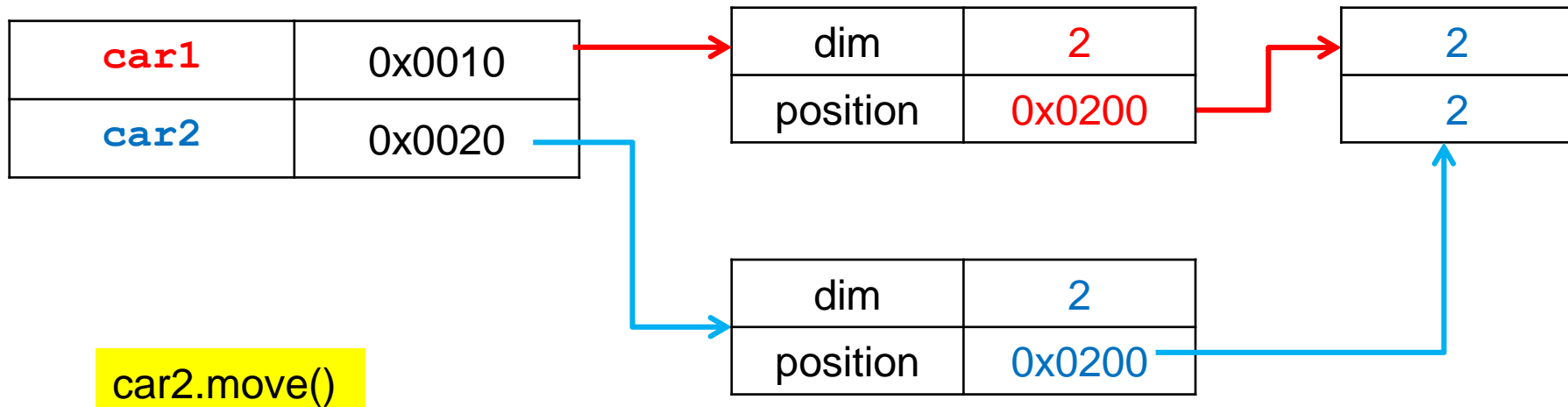
Ρηχά Αντίγραφα

- Η copy όπως την έχουμε ορίσει δημιουργεί ένα **ρηχό αντίγραφο** του αντικειμένου
 - Αντιγράφει τις **αναφορές** στα αντικείμενα και όχι τα **περιεχόμενα** των αντικειμένων



Ρηχά Αντίγραφα

- Η copy όπως την έχουμε ορίσει δημιουργεί ένα **ρηχό αντίγραφο** του αντικειμένου
 - Αντιγράφει τις **αναφορές** στα αντικείμενα και όχι τα **περιεχόμενα** των αντικειμένων

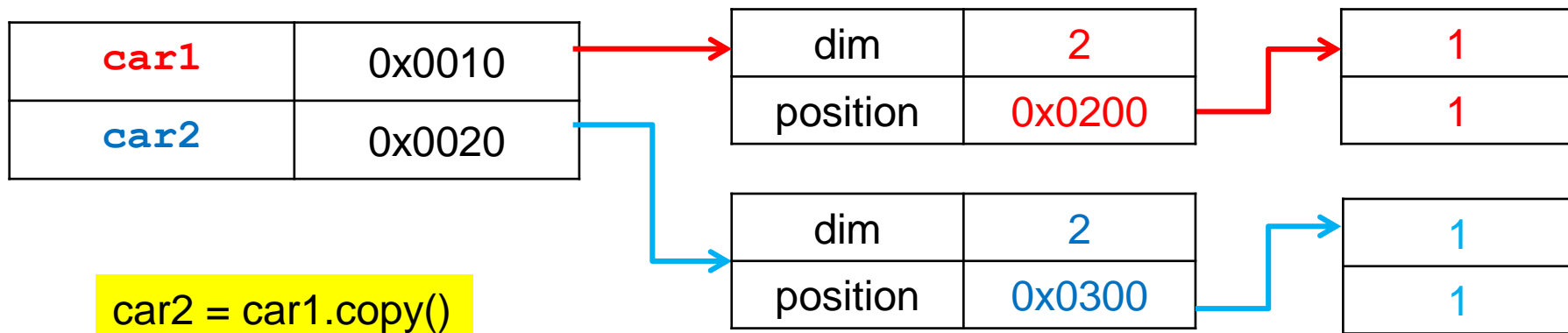


Μετακινείται και το car1 αλλά αυτό δεν είναι επιθυμητό.

Βαθύ αντίγραφο

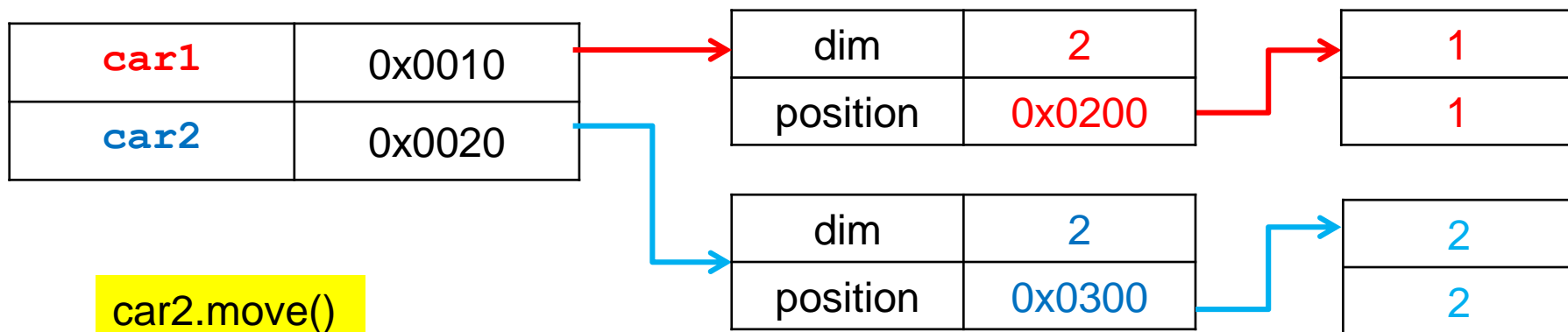
- Τις περισσότερες φορές θέλουμε να κάνουμε ένα **βαθύ αντίγραφο** του αντικειμένου, όπου για κάθε αντικείμενο μέσα στο αντίγραφο δεσμεύουμε νέα μνήμη

```
public Car copy() {  
    Car newCar = new Car(this.dim);  
    for (int i=0; i<dim; i++){  
        newCar.position[i] = this.position[i];  
    }  
    return newCar;  
}
```



Βαθύ αντίγραφο

- Το **βαθύ αντίγραφο** του car1 είναι πλέον ένα ανεξάρτητο αντικείμενο.



Η μετακίνηση του car2 δεν επηρεάζει το car1

Copy Constructor

- Ένας Constructor που παίρνει σαν όρισμα ένα αντικείμενο του ίδιου τύπου και δημιουργεί ένα αντίγραφο
 - `public Car (Car other)`
- Ο `copy constructor` έχει δύο λειτουργίες:
 - **Δεσμεύει** τη μνήμη για το αντικείμενο
 - **Αντιγράφει** τις τιμές του αντικειμένου-ορίσματος.
- **Πάντα** πρέπει να δημιουργούμε ένα **βαθύ αντίγραφο** του αντικειμένου

Copy Constructor για την Car

```
public Car(Car other)
{
    this.dim = other.dim;
    position = new int[this.dim];
    for (int i = 0; i < this.dim; i ++){
        this.position[i] = other.position[i];
    }
}
```

Δημιουργεί **βαθύ αντίγραφο**:

Δεσμεύουμε καινούριο πίνακα και αντιγράφουμε μία-μία τις τιμές

Κλήση:

```
Car car1 = new Car(2);
```

```
Car car2 = new Car(car1);
```

Φωλιασμένος Copy Constructor

- Αν μια κλάση έχει **πεδία αντικείμενα** από μία **άλλη κλάση**, τότε όταν καλούμε τον copy constructor θα πρέπει να έχουμε ορίσει **copy constructor** και για τις κλάσεις των αντικειμένων-πεδίων.

Παράδειγμα

```
class CarDriver
{
    private int position;
    private Person driver;

    public CarDriver(CarDriver other) {
        this.position = other.position;
        driver = new Person(other.driver);
    }
}
```

Καλεί την **copy constructor** της **Person**

```
class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber) {
        name = initName; number = initNumber;
    }

    public Person(Person other) {
        this.name = other.name;
        this.number = other.number;
    }

    public void set(String newName, int newNumber) {
        name = newName;
        number = newNumber;
    }

    public String toString() {
        return (name + " " + number);
    }

    public boolean equals(Person other) {
        return (this.name.equals(other.name) && this.number == other.number);
    }
}
```

Η ΑΝΑΦΟΡΑ ΤΗΣ

Η μεταβλητή **this**

- Η μεταβλητή (παράμετρος) **this**
 - Μια **κρυφή παράμετρος** η οποία περνάει σε κάθε μέθοδο και κρατάει μια αναφορά στο **αντικείμενο κλήσης** (το αντικείμενο που καλεί την μέθοδο).
- Την χρησιμοποιήσαμε για να διαφοροποιήσουμε τα πεδία του αντικειμένου από παραμέτρους με το ίδιο όνομα

```
class Person
{
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Η μεταβλητή **this**

- Εκτός από αυτή την χρήση, μπορούμε να χρησιμοποιήσουμε την μεταβλητή **this** σαν οποιαδήποτε άλλη μεταβλητή
 - Μπορούμε να την **επιστρέψουμε** σε κάποια μέθοδο.
 - Μπορούμε να την **αναθέσουμε** σε κάποια μεταβλητή
 - Μπορούμε να την **περάσουμε σαν παράμετρο** σε κάποια μέθοδο
- Αυτό είναι χρήσιμο όταν χρειαζόμαστε μια αναφορά στο αντικείμενο κλήσης.


```
class Person
```

```
{
```

```
    private String name;
```

```
    private int age;
```

```
    private Person spouse;
```

```
    public Person(String name, int age) {
```

```
        this.name = name;
```

```
        this.age = age;
```

```
    }
```

```
    public Person getOlderPerson(Person other) {
```

```
        if (this.age > other.age) {
```

```
            return this;
```

```
        }
```

```
        return other;
```

```
    }
```

```
    public void marry(Person other) {
```

```
        this.spouse = other;
```

```
        other.spouse = this;
```

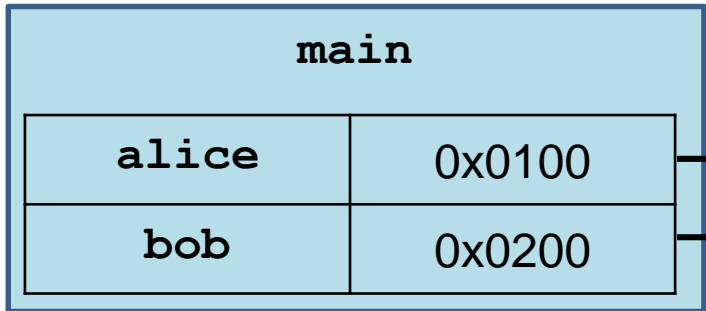
```
    }
```

```
}
```

Η μέθοδος μας επιστρέφει μια αναφορά σε αντικείμενο Person
Αν η ηλικία του ατόμου που καλεί την μέθοδο είναι μεγαλύτερη αυτού που περνάμε σαν όρισμα επιστρέφουμε την αναφορά **this**
Αλλιώς επιστρέφουμε την αναφορά **other**.

Θέτουμε τον/την σύζυγο του other να δείχνει στο αντικείμενο **this**

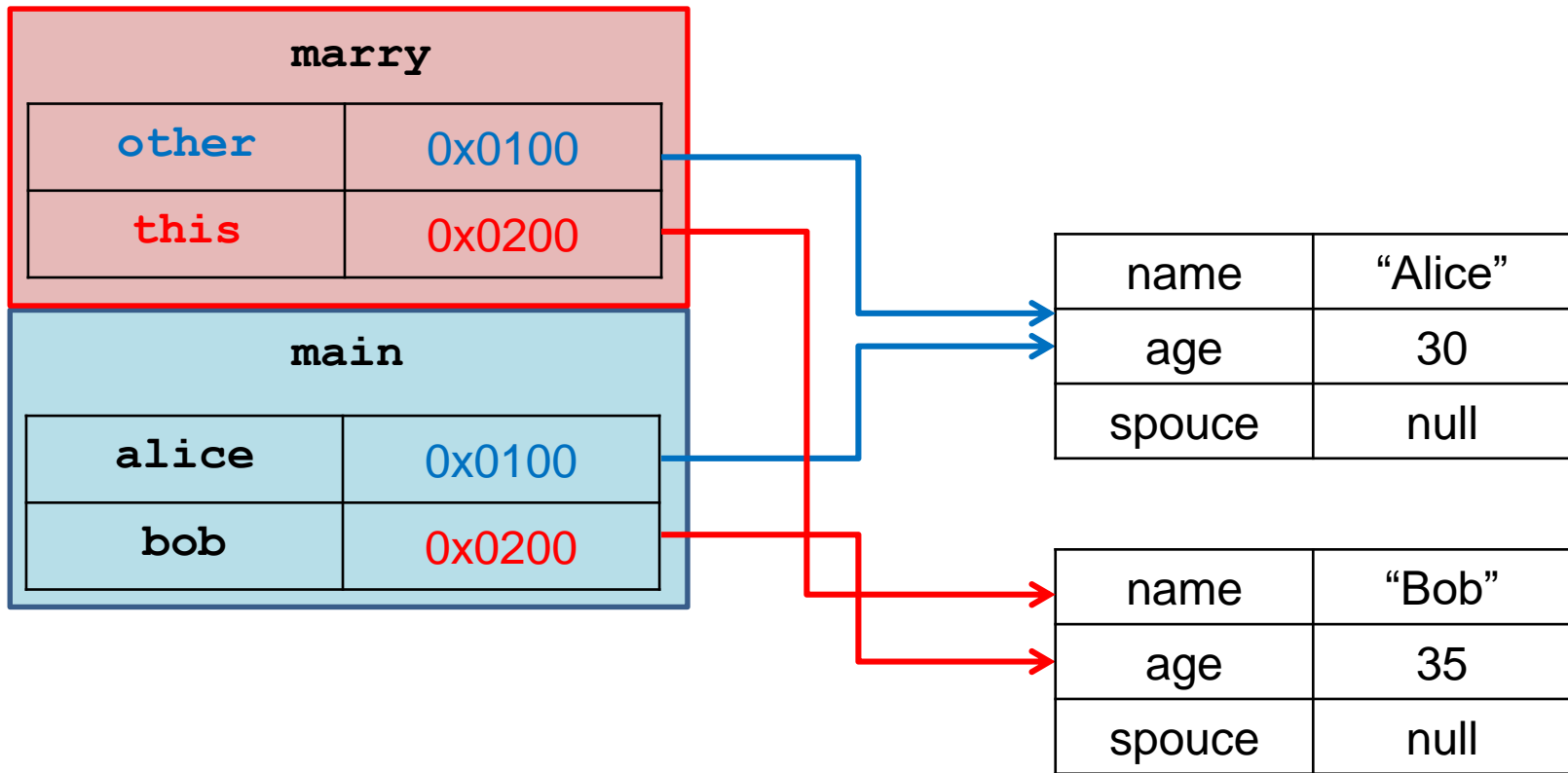
```
class MarriageTest1
{
    public static void main(String[] args){
        Person alice = new Person("Alice", 30);
        Person bob = new Person("Bob", 35);
        Person older = bob.getOlderPerson(alice);
        bob.marry(alice);
    }
}
```



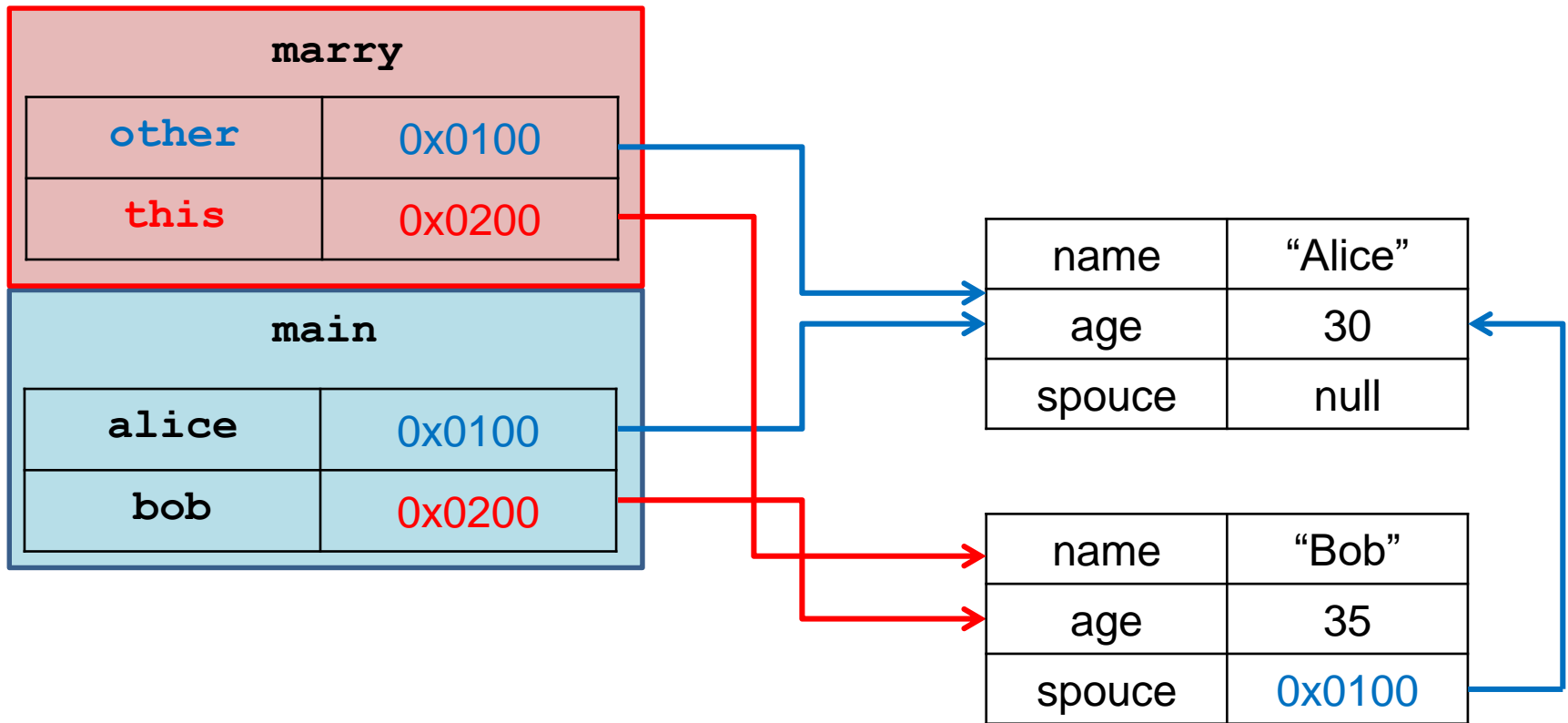
name	"Alice"
age	30
spouce	null

name	"Bob"
age	35
spouce	null

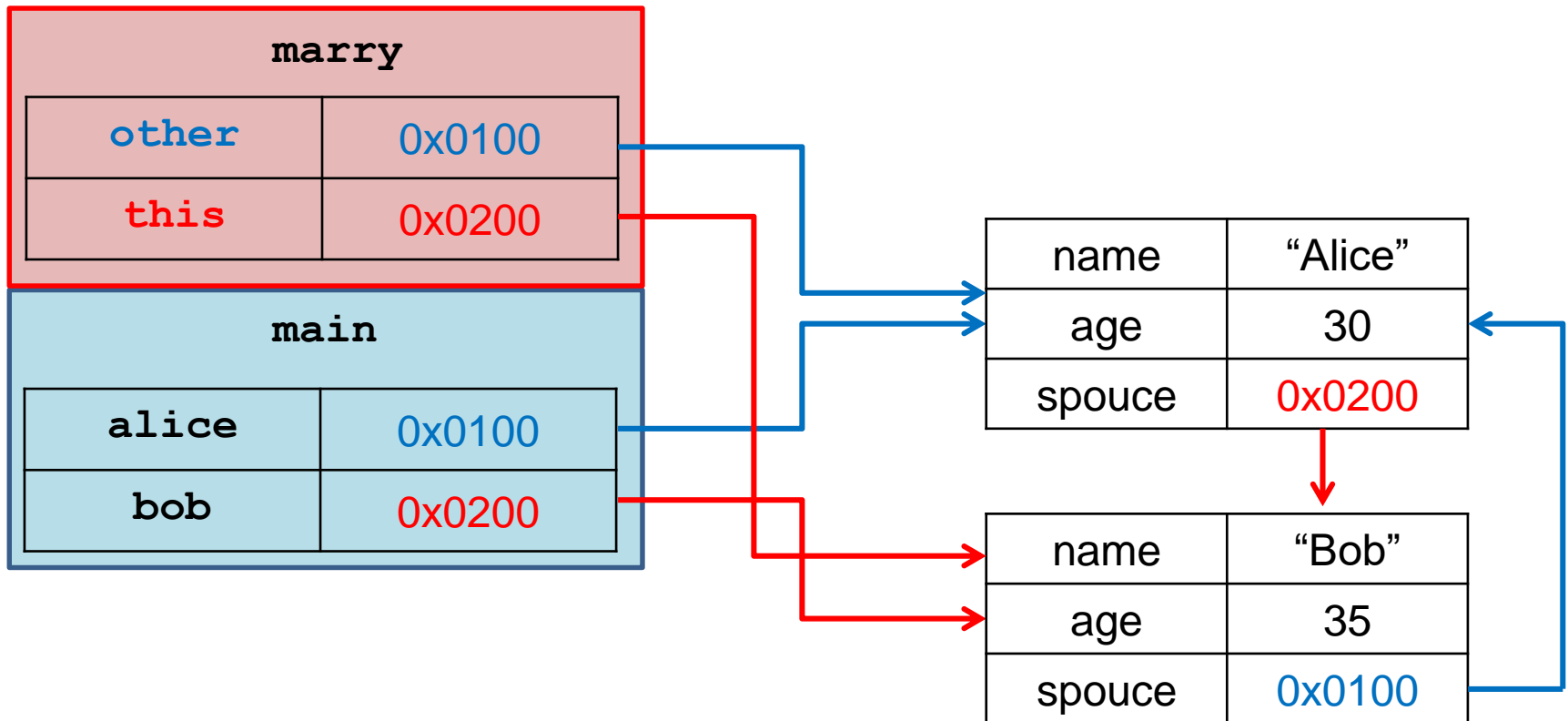
```
bob.marry(alice);
```



```
public void marry(Person other) {  
    this.spouce = other;  
    other.spouce = this;  
}
```



```
public void marry(Person other) {  
    this.spouce = other;  
    other.spouce = this;  
}
```



Παρένθεση: Τα περιεχόμενα ενός ArrayList μπορεί να είναι οτιδήποτε, π.χ., στην περίπτωση αυτή πίνακες από αντικείμενα Person

```
class Registry
{
    private ArrayList<Person[]> marriages =
        new ArrayList<Person[]>();

    public void addMariage(Person A, Person B) {
        Person[] couple = new Person[2];
        couple[0] = A;
        couple[1] = B;
        marriages.add(couple);
    }
}
```

```
class Person
```

```
{
```

```
    private String name;
```

```
    private int age;
```

```
    private Person spouse;
```

```
    public Person(String name, int age) {
```

```
        this.name = name;
```

```
        this.age = age;
```

```
    }
```

```
    public Person getOlderPerson(Person other) {
```

```
        if (this.age > other.age) {
```

```
            return this;
```

```
        }
```

```
        return other;
```

```
    }
```

```
    public void marry(Person other, Registry reg) {
```

```
        this.spouse = other;
```

```
        other.spouse = this;
```

```
        reg.addMarriage(this, other)
```

```
    }
```

```
}
```

Προσθέτουμε ένα νέο ζευγάρι με τα αντικείμενα **this** και **other**


```
class MarriageTest2
{
    public static void main(String[] args){
        Person alice = new Person("Alice", 30);
        Person bob = new Person("Bob", 35);
        Registry cityReg = new Registry();
        Person older = bob.getOlderPerson(alice);
        bob.marry(alice, cityReg);
    }
}
```

ΣΥΝΘΕΣΗ ΚΑΙ ΑΝΑΦΟΡΕΣ

Σύνθεση

- Μια κλάση μπορεί να χρησιμοποιεί αντικείμενα άλλων κλάσεων.
 - Αυτή η διαδικασία ονομάζεται **σύνθεση**.
- Όταν έχουμε σύνθεση κλάσεων πρέπει να είμαστε προσεκτικοί για το πώς μεταβάλλονται τα περιεχόμενα αντικειμένων στα οποία δείχνουν πολλαπλές αναφορές.

Παραδείγματα

- Τι γίνεται αν έχουμε ένα constructor που παίρνει όρισμα ένα πίνακα?
 - `public Car(int[] position)`
 - Αν ο πίνακας αλλάξει μέσα στην main θα αλλάξει και στο αντικείμενο.
- Τι γίνεται αν στο κάνουμε τον πίνακα null στην main?

```

class CarArray
{
    private int[] position;
    private int dim;

    public CarArray(int[] array){
        dim = array.length;
        position = array;
    }

    public void move(){
        for (int i=0; i < dim; i++){
            position[i] ++;
        }
    }

    public String toString(){
        String output = "";
        for (int i=0; i < dim; i++){
            output = output + position[i] + " ";
        }
        return output;
    }

    public static void main(String args[]){
        int[] pos = {1,2};
        CarArray myCar = new CarArray(pos);
        myCar.move(); System.out.println(pos[0]+ " " + pos[1]);
        pos[0] ++ ; System.out.println(myCar);
        pos = null; System.out.println(myCar);
    }
}

```

Τι θα τυπώσει?

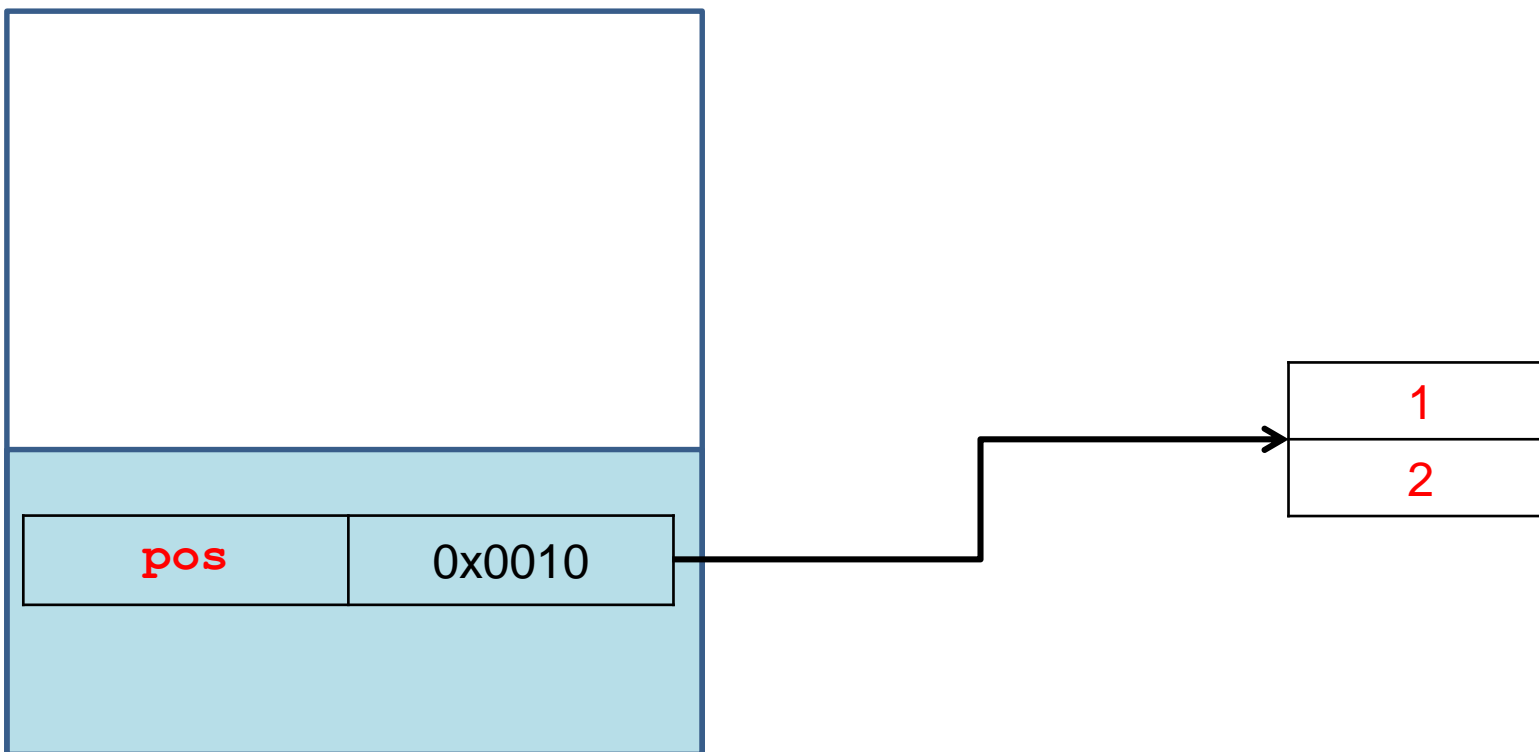
Η αλλαγή στα περιεχόμενα του πίνακα στο αντικείμενο myCar αλλάζει και τα περιεχόμενα του pos

Η αλλαγή στην τιμή του pos **δεν** αλλάζει τα περιεχόμενα το πεδίο στο αντικείμενο myCar

Η αλλαγή στα περιεχόμενα του pos αλλάζει και τα περιεχόμενα του πίνακα στο αντικείμενο myCar

Εκτέλεση

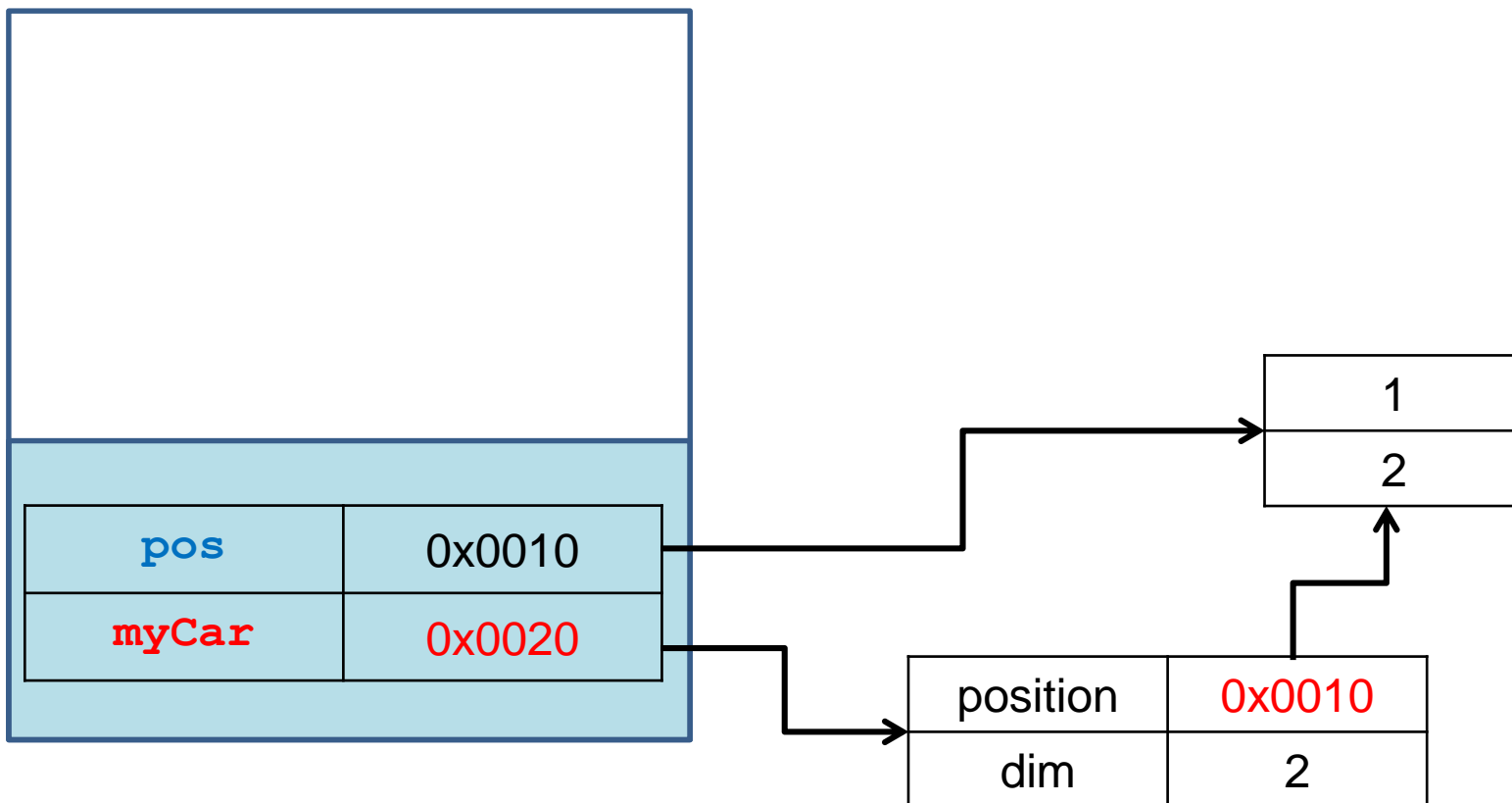
```
int[] pos = {1,2}
```



Εκτέλεση

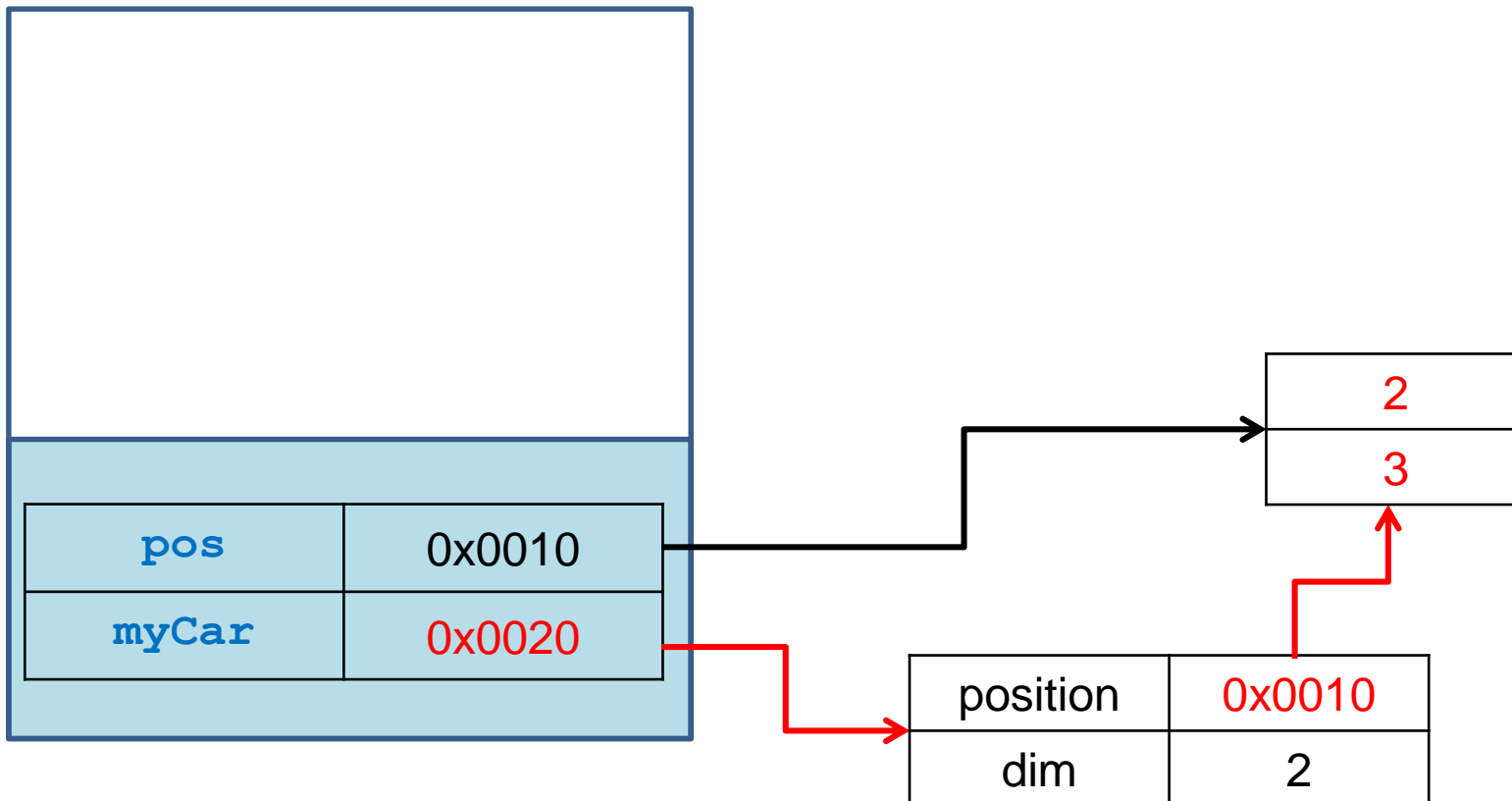
```
int[] pos = {1,2}
```

```
CarArray myCar = new CarArray(pos);
```



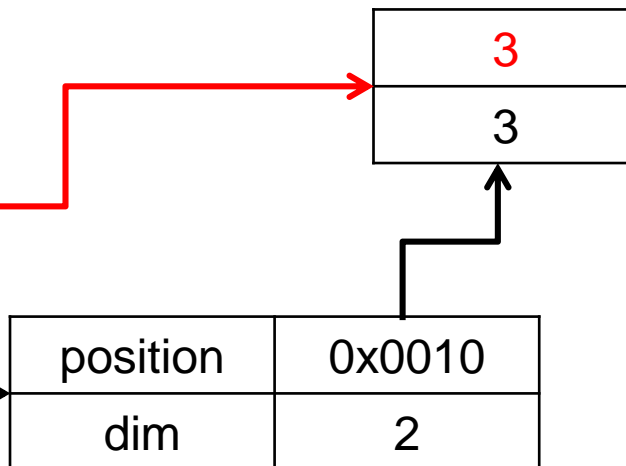
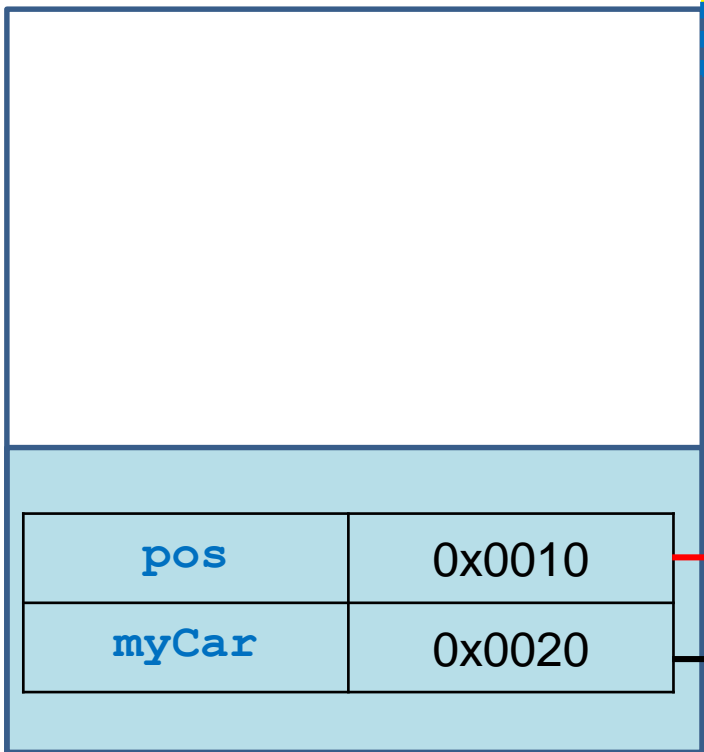
Εκτέλεση

```
int[] pos = {1,2}  
CarArray myCar = new CarArray(pos);  
myCar.move();  
System.out.println(pos[0]+" "+pos[1]);
```



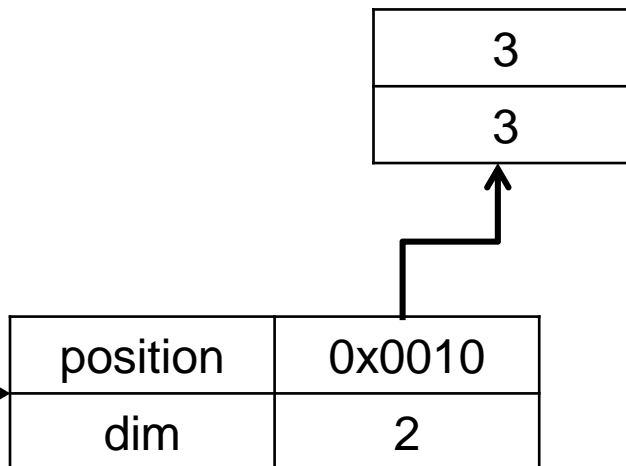
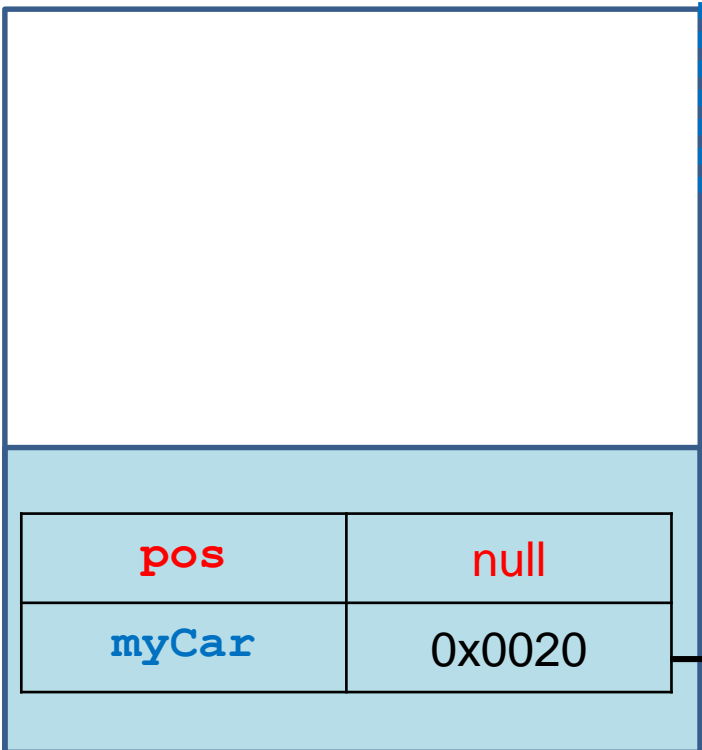
Εκτέλεση

```
int[] pos = {1,2}
CarArray myCar = new CarArray(pos);
myCar.move();
System.out.println(pos[0]+" "+pos[1]);
pos[0] ++ ;
System.out.println(myCar);
```



Εκτέλεση

```
int[] pos = {1,2}  
CarArray myCar = new CarArray(pos);  
myCar.move();  
System.out.println(pos[0]+" "+pos[1]);  
pos[0] ++;  
System.out.println(myCar);  
pos = null;  
System.out.println(myCar);
```



```
class Person
{
    private String name;

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

    public String getName(){
        return name;
    }

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

```
class Car
{
    private int position = 0;
    private Person driver;

    public Car(int position, Person driver){
        this.position = position;
        this.driver = driver;
    }

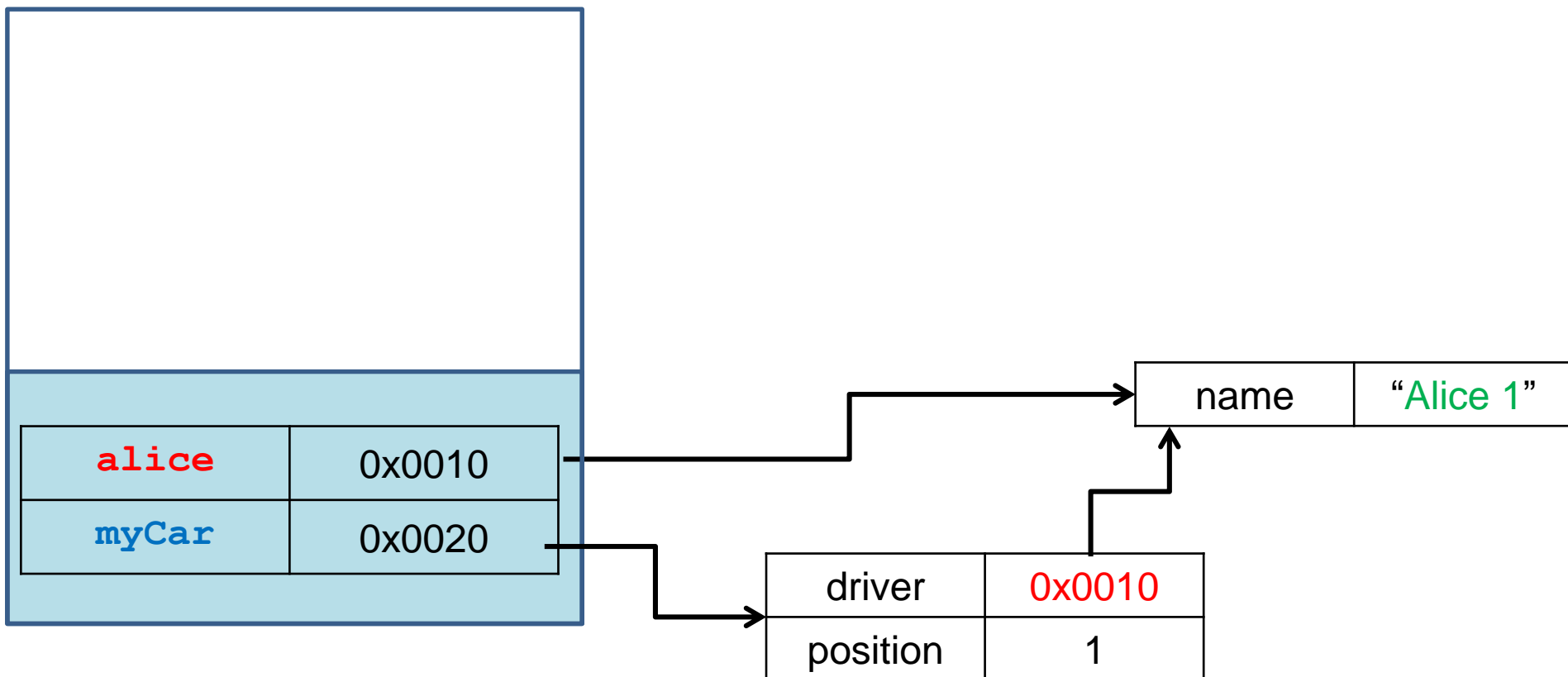
    public Person getDriver(){
        return driver;
    }
}
```

```
class MovingCarDriver3
{
    public static void main(String args[]){
        Person alice = new Person("Alice 1");
        Car myCar = new Car(1, alice);
        alice.setName("Alice 2");
        System.out.println(myCar.getDriver().getName());
        alice = new Person("Alice 3");
        System.out.println(myCar.getDriver().getName());
    }
}
```

Τι θα τυπώσει?

Εκτέλεση

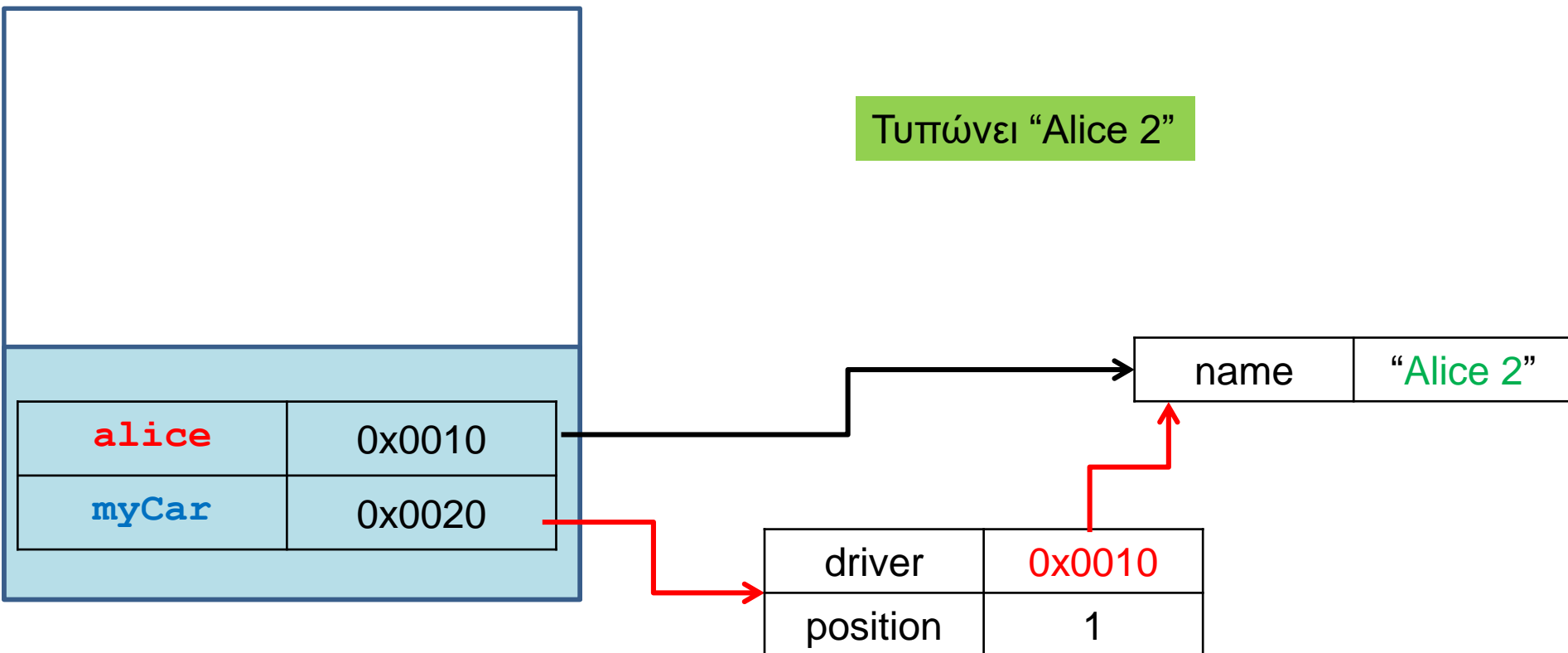
```
Person alice = new Person("Alice 1");  
Car myCar = new Car(1, alice);
```



Εκτέλεση

```
alice.setName("Alice 2");  
System.out.println(myCar.getDriver().getName());
```

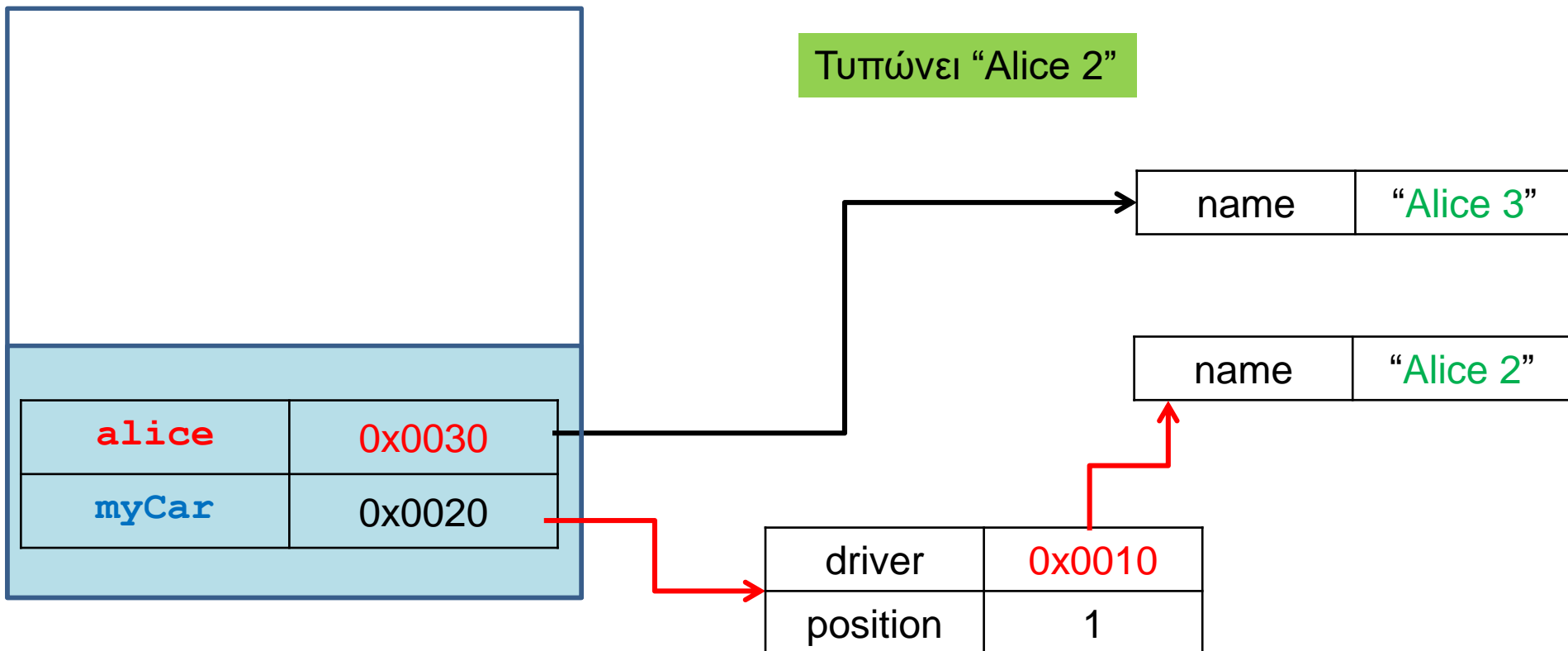
Τυπώνει "Alice 2"



Εκτέλεση

```
alice = new Person("Alice 3");  
System.out.println(myCar.getDriver().getName());
```

Τυπώνει "Alice 2"



Αντικείμενα μέσα σε αντικείμενα

- Ορίζουμε κλάσεις για να ορίσουμε **τύπους δεδομένων** τους οποίους χρειαζόμαστε
 - Π.χ., ο τύπος δεδομένων **Person** για να μπορούμε να χειριζόμαστε πληροφορίες για ένα άτομο, και ο τύπος δεδομένων **Car** που κρατάει πληροφορία για το αυτοκίνητο.
- Τους τύπους δεδομένων που ορίζουμε τους χρησιμοποιούμε για να δημιουργήσουμε **μεταβλητές** (αντικείμενα).
- Τα αντικείμενα μπορεί να είναι **πεδία** άλλων κλάσεων
 - Π.χ., η κλάση Car έχει ένα πεδίο τύπου Person
- Μία κλάση χρησιμοποιεί αντικείμενα άλλων κλάσεων και έτσι **συνθέτουμε** πιο περίπλοκους τύπους δεδομένων.

7. ΣΥΝΘΕΣΗ

Σύνθεση και συνάθροιση

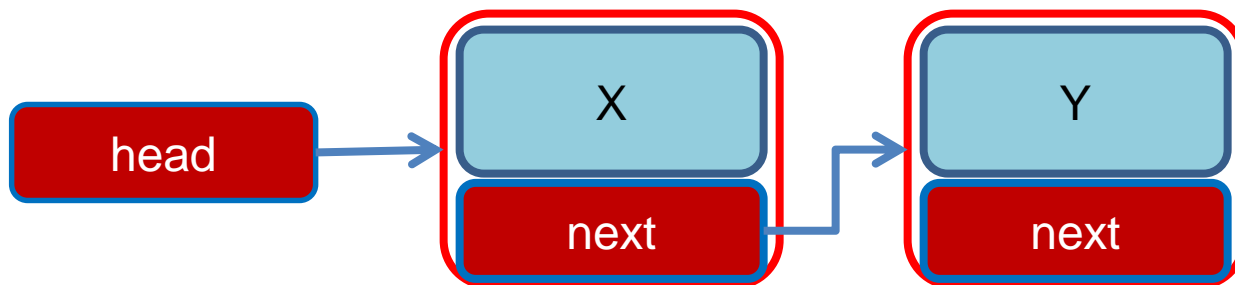
Υλοποίηση δυναμικής στοίβας

Παράδειγμα: Τμήμα Πανεπιστημίου

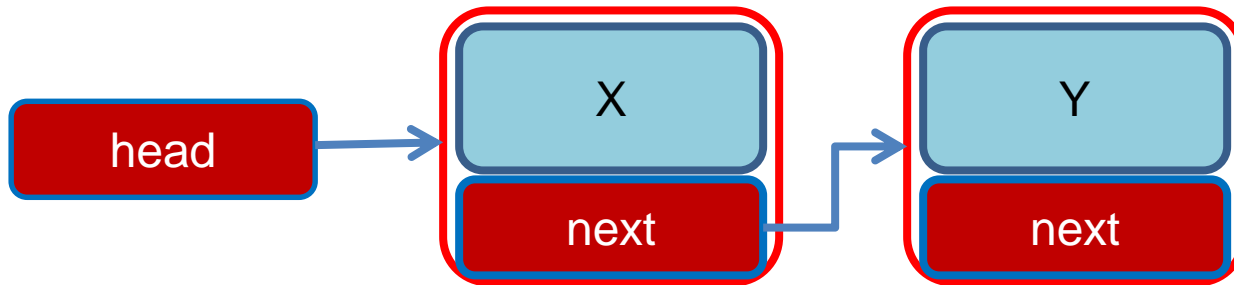
ΥΛΟΠΟΙΗΣΗ ΔΥΝΑΜΙΚΗΣ ΣΤΟΙΒΑΣ

Παράδειγμα

- Υλοποιήστε το Stack που φτιάξαμε στα προηγούμενα μαθήματα ώστε να μην έχει περιορισμό στο μέγεθος (capacity).
- Βασική ιδέα:
 - Δημιουργούμε στοιχεία της στοίβας και τα συνδέουμε το ένα να δείχνει στο άλλο.
 - Χρειάζεται να ξέρουμε και την κορυφή της στοίβας.

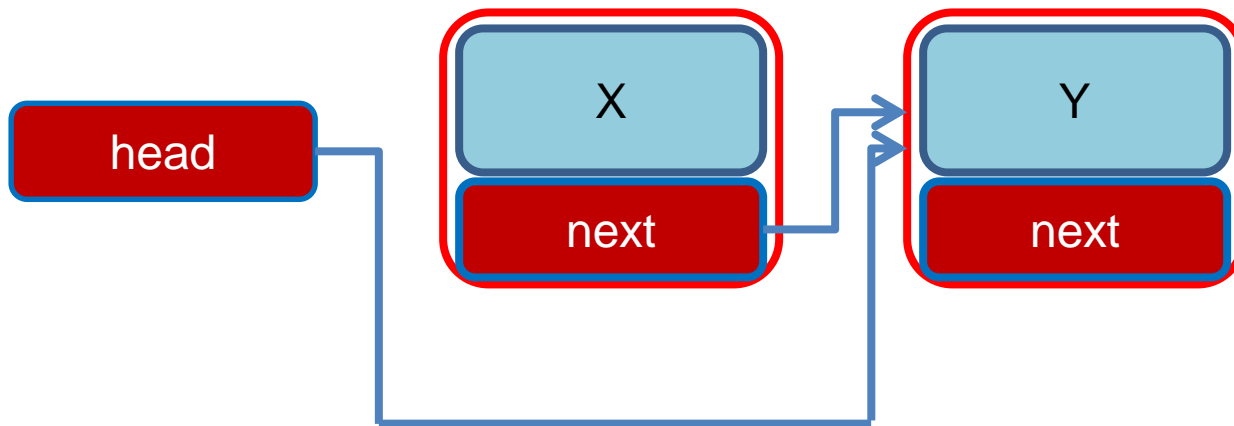


Στοίβα



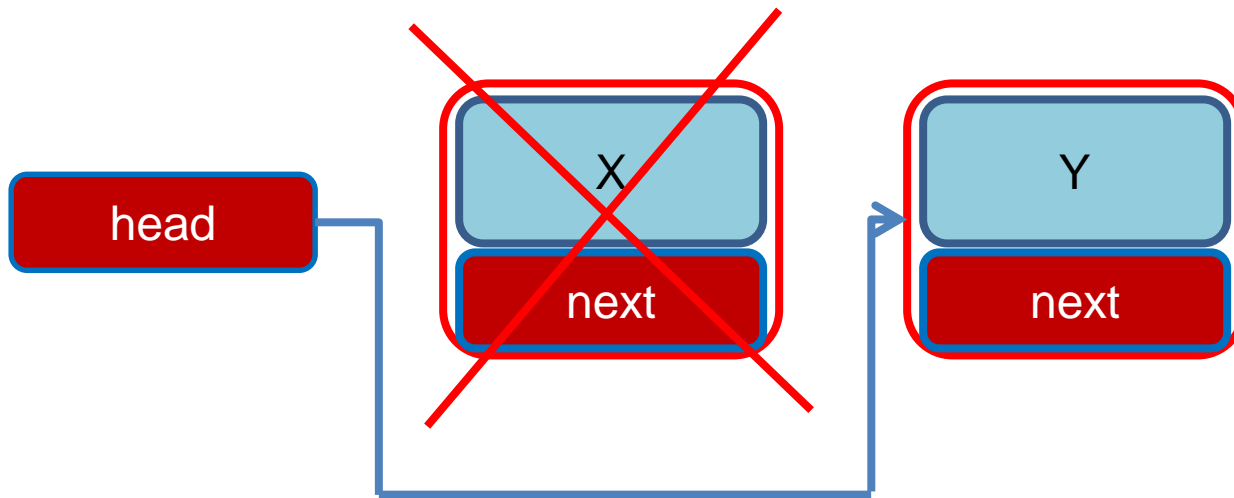
Pop(): Αφαιρεί το στοιχείο στην κορυφή της στοίβας και επιστρέφει την τιμή του (X στο παράδειγμα μας)

Στοίβα



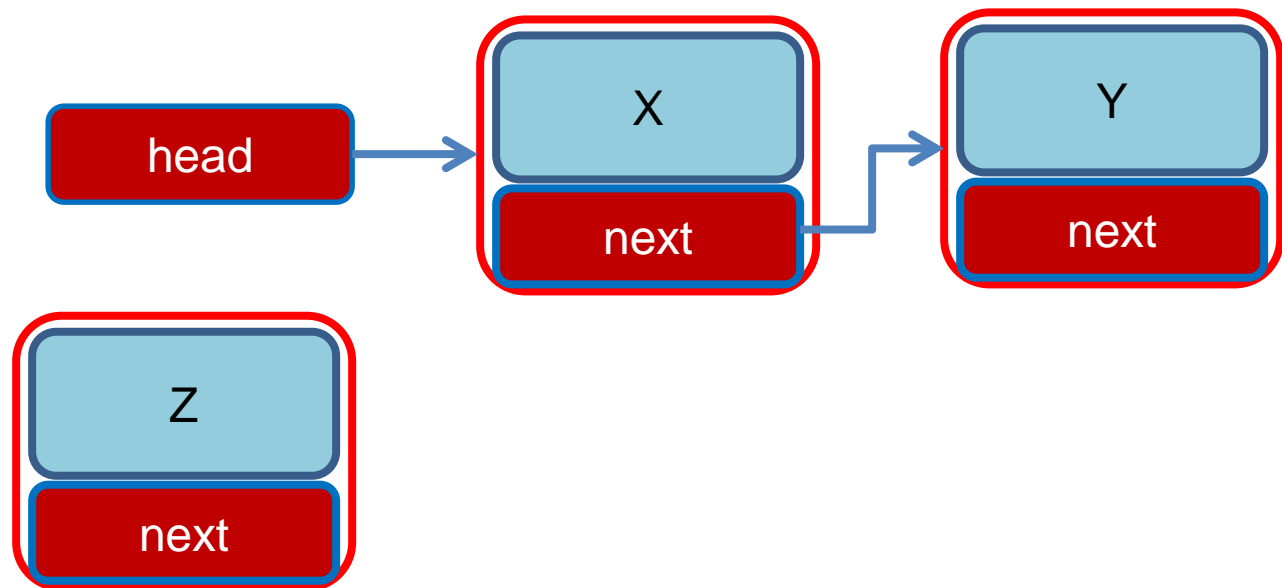
Pop(): Αφαιρεί το στοιχείο στην κορυφή της στοίβας και επιστρέφει την τιμή του (X στο παράδειγμα μας)

Στοίβα



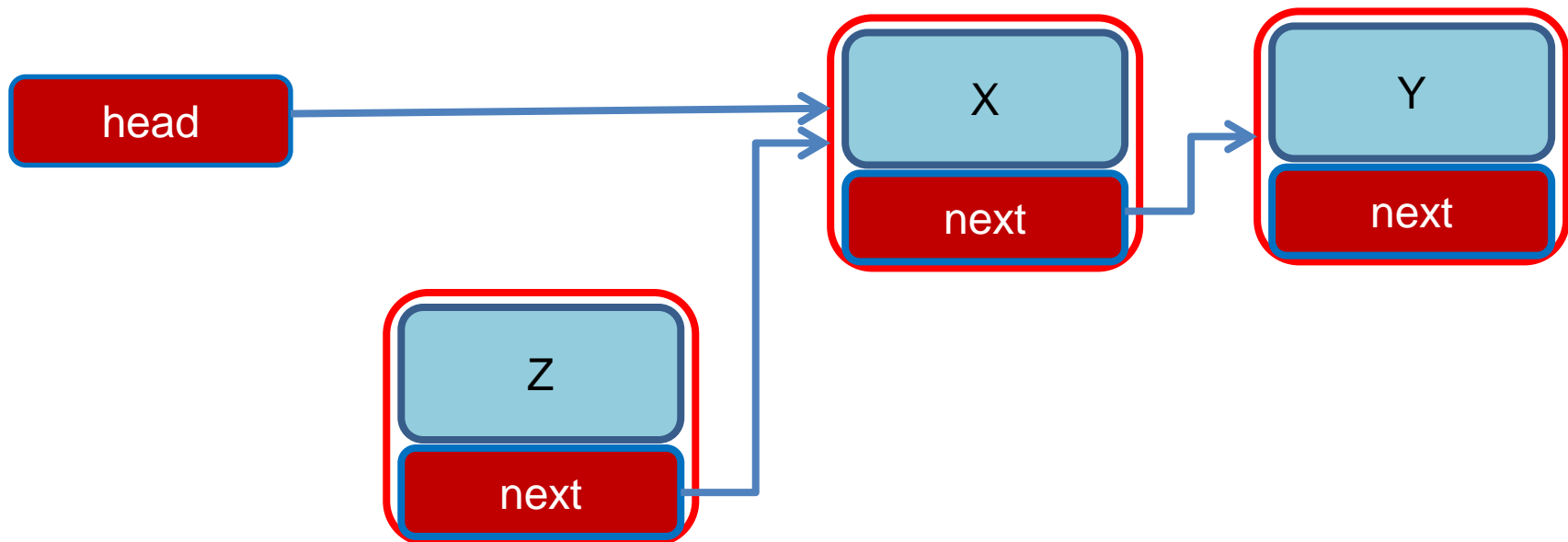
Pop(): Αφαιρεί το στοιχείο στην κορυφή της στοίβας και επιστρέφει την τιμή του (X στο παράδειγμα μας)

Στοιίβα



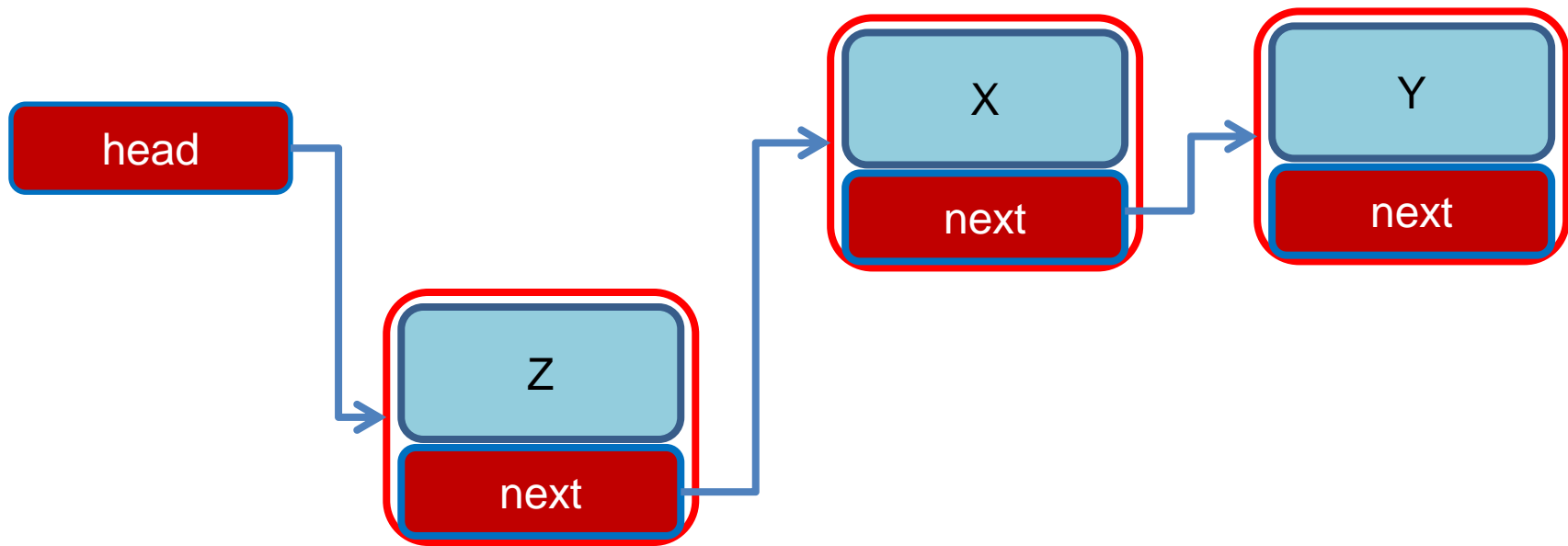
Push(Z): Προσθέτει την τιμή Z στην κορυφή της στοίβας

Στοίβα



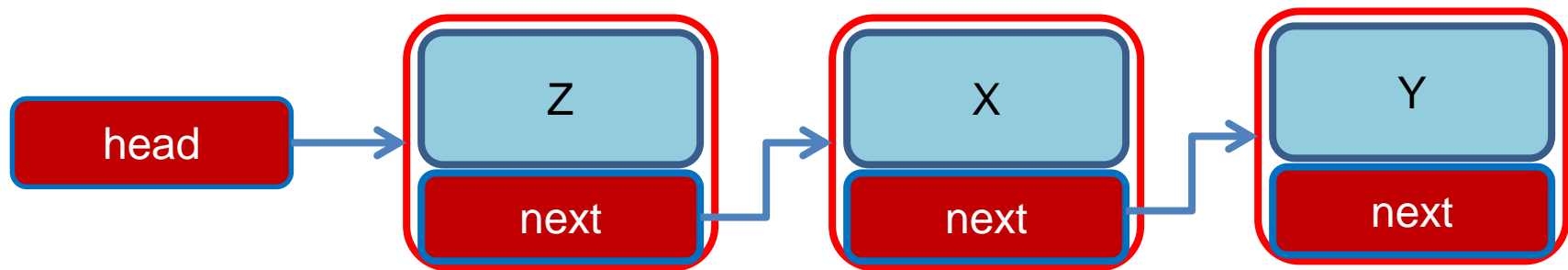
Push(Z): Προσθέτει την τιμή Z στην κορυφή της στοίβας

Στοιίβα



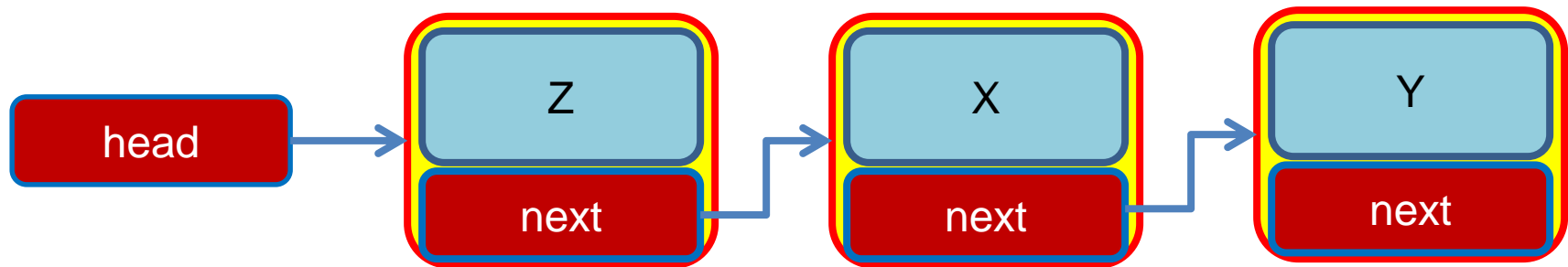
Push(Z): Προσθέτει την τιμή Z στην κορυφή της στοίβας

Στοίβα



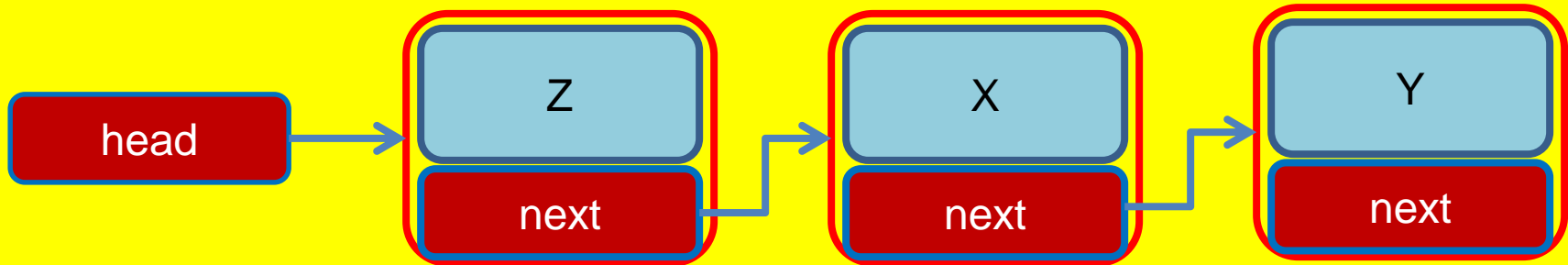
Push(Z): Προσθέτει την τιμή Z στην κορυφή της στοίβας

Στοιίβα - Υλοποίηση



- Θα ορίσουμε **StackElement** μια κλάση που κρατάει το κάθε στοιχείο της στοίβας.

Στοίβα - Υλοποίηση



- Θα ορίσουμε **StackElement** μια κλάση που κρατάει το κάθε στοιχείο της στοίβας.
- Και μια κλάση **Stack** που υλοποιεί την στοίβα και όλες τις λειτουργίες της

```
class StackElement
```

```
{
```

```
    private int value;
```

```
    private StackElement next = null;
```

```
    public StackElement(int value){
```

```
        this.value = value;
```

```
    }
```

```
    public int getValue(){
```

```
        return value;
```

```
    }
```

```
    public StackElement getNext(){
```

```
        return next;
```

```
    }
```

```
    public void setNext(StackElement element){
```

```
        next = element;
```

```
    }
```

```
}
```

Το επόμενο στοιχείο

Επιστρέφει αντικείμενο

```
class Stack
```

```
{
```

```
    private StackElement head;
```

```
    private int size = 0;
```

```
    public int pop(){
```

```
        if (size == 0){ // head == null
```

```
            System.out.println("Pop from empty stack");
```

```
            System.exit(-1);
```

```
        }
```

```
        int value = head.getValue();
```

```
        head = head.getNext();
```

```
        size --;
```

```
        return value;
```

```
    }
```

```
    public void push(int value){
```

```
        StackElement element = new StackElement(value);
```

```
        element.setNext(head);
```

```
        head = element;
```

```
        size ++;
```

```
    }
```

```
}
```

Το πρώτο στοιχείο της στοίβας μας φτάνει για τα βρούμε όλα

Σταματάει την εκτέλεση του προγράμματος

Τα αντικείμενα τύπου StackElement δημιουργούνται μέσα στην Stack.

Μέθοδος toString()

```
public String toString(){
    String returnString = "";
    IntStackElement e = head;
    for (int i = 0; i < size; i ++){
        returnString = returnString + e.getValue() + " ";
        e = e.getNext();
    }
    return returnString;
}
```

Χρειαζόμαστε μία StackElement μεταβλητή για να διατρέξει τα στοιχεία της στοίβας

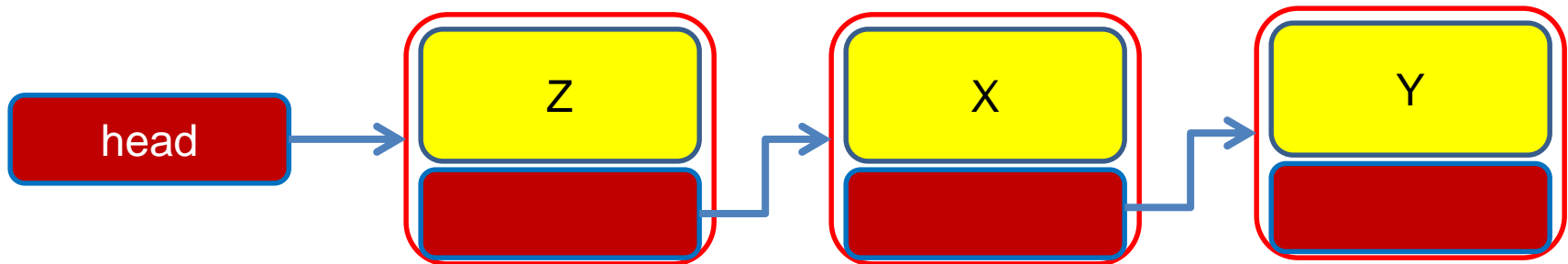
```
public String toString(){
    String returnString = "";
    for (IntStackElement e = head;
        e != null;
        e = e.getNext()){
        returnString = returnString + e.getValue() + " ";
    }
    return returnString;
}
```

Εναλλακτική υλοποίηση

for-loop που δεν διατρέχει ακεραίους

```
class StackExample
{
    public static void main(String[] args) {
        Stack s = new Stack();
        s.push(3);
        s.push(2);
        s.push(1);
        System.out.println(s.pop());
        System.out.println(s.pop());
        System.out.println(s.pop());
        System.out.println(s.pop());
    }
}
```

Στοίβα - Υλοποίηση



- Τα X, Y, Z μπορεί να είναι δεδομένα οποιουδήποτε τύπου ή κλάσης. Π.χ. αντί για ακέραιους θα μπορούσαμε να έχουμε αντικείμενα τύπου **Person**.


```
class Person
{
    private String name;
    private int number;

    public Person(String name, int num){
        this.name = name;
        this.number = num;
    }

    public String toString(){
        return name+": "+number;
    }
}
```

```
class PersonStackElement
{
    private Person value;
    private PersonStackElement next;

    public PersonStackElement(Person val) {
        value = val;
    }

    public void setNext(PersonStackElement element) {
        next = element;
    }

    public PersonStackElement getNext() {
        return next;
    }

    public Person getValue() {
        return value;
    }
}
```

Ο constructor παίρνει σαν όρισμα το αντικείμενο που έχει ήδη δημιουργηθεί

Το αντικείμενο το χειριζόμαστε σαν μια οποιαδήποτε μεταβλητή

```
class Stack
```

```
{
```

```
    private PersonStackElement head;  
    private int size = 0;
```

Η pop πλέον επιστρέφει μεταβλητή
τύπου Person

```
    public Person pop() {
```

```
        if (size == 0) { // head == null
```

```
            System.out.println("Pop from empty stack");
```

```
            return null;
```

```
        }
```

```
        int value = head.getValue();
```

```
        head = head.getNext();
```

```
        size --;
```

```
        return value;
```

```
    }
```

Επιστρέφουμε null για να
σηματοδοτήσουμε ότι έγινε λάθος
(όχι απαραίτητα ο καλύτερος
τρόπος να το κάνουμε αυτό)

```
    public void push(Person value) {
```

```
        StackElement element = new StackElement(value);
```

```
        element.setNext(head);
```

```
        head = element;
```

```
        size ++;
```

```
    }
```

```
}
```

```
class StackExample
{
    public static void main(String[] args){
        PersonStack stack = new PersonStack();
        Person alice = new Person("Alice", 1);
        stack.push(alice);
        Person bob = new Person("Bob", 2);
        stack.push(bob);
        Person charlie = new Person("Charlie", 3);
        stack.push(charlie);
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

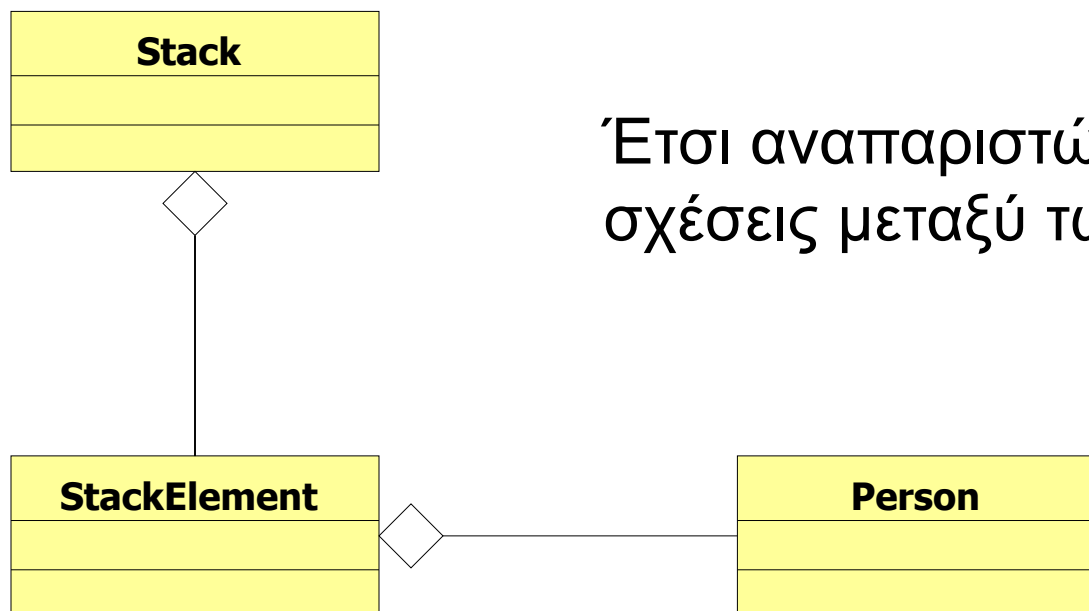
Προσοχή! Αν καλέσουμε άλλη μια φορά την pop θα πάρουμε runtime error γιατί προσπαθούμε να προσπελάσουμε null αναφορά

Σχέσεις μεταξύ κλάσεων

- Στο παράδειγμα με τη στοίβα έχουμε τρεις διαφορετικές κλάσεις (**Person**, **StackElement**, **Stack**) τις οποίες συσχετίζονται μεταξύ τους με διαφορετικούς τρόπους.
- Μπορεί να υπάρχουν πολλές διαφορετικές σχέσεις μεταξύ κλάσεων.
 - Στην περίπτωση μας, η μία κλάση ορίζεται χρησιμοποιώντας αντικείμενα της άλλης
- Αυτού του είδους τη σχέση την λέμε σχέση **σύνθεσης**
 - Μερικές φορές την ξεχωρίζουμε σε σχέση **σύνθεσης** (composition) και **συνάθροισης** (aggregation).

Η UML γλώσσα

- Η **UML (Unified Modeling Language)** είναι μια γλώσσα για να περιγράψουμε και να καταλαβαίνουμε τον κώδικα μας.
- Τα **UML διαγράμματα** παρέχουν μια οπτικοποίηση των σχέσεων μεταξύ των κλάσεων.



Έτσι αναπαριστώνται οι σχέσεις μεταξύ των κλάσεων

Σχέσεις κλάσεων

- Όταν έχουμε **κλάσεις** που **έχουν αντικείμενα άλλων κλάσεων** ένα θέμα που προκύπτει είναι πότε και πού θα γίνεται η **δημιουργία των αντικειμένων** και πότε η καταστροφή τους
 - Πιο σημαντικό σε γλώσσες που δεν έχουν garbage collector.
- Π.χ., τα αντικείμενα τύπου **StackElement** στο προηγούμενο παράδειγμα **δημιουργούνται μέσα** στην κλάση **Stack**, και καταστρέφονται μέσα στην **Stack**, ή αν η **Stack** καταστραφεί.
 - Αλλαγές σε **StackElement** αντικείμενα γίνονται **μόνο** μέσα στην **Stack**
- Τα αντικείμενα τύπου **Person** που χρησιμοποιούνται στην **StackElement** **δημιουργούνται εκτός της κλάσης** και μπορεί να υπάρχουν αφού καταστραφεί η κλάση.
 - Αλλαγές στα αντικείμενα **Person** επηρεάζουν και τα περιεχόμενα της **Stack** και τούμπαλιν.
- Συχνά οι σχέσεις του δεύτερου τύπου λέγονται σχέσεις **συνάθροισης**, ενώ του πρώτου σχέσεις **σύνθεσης**.

Σχέση συνάθροισης – Aggregation

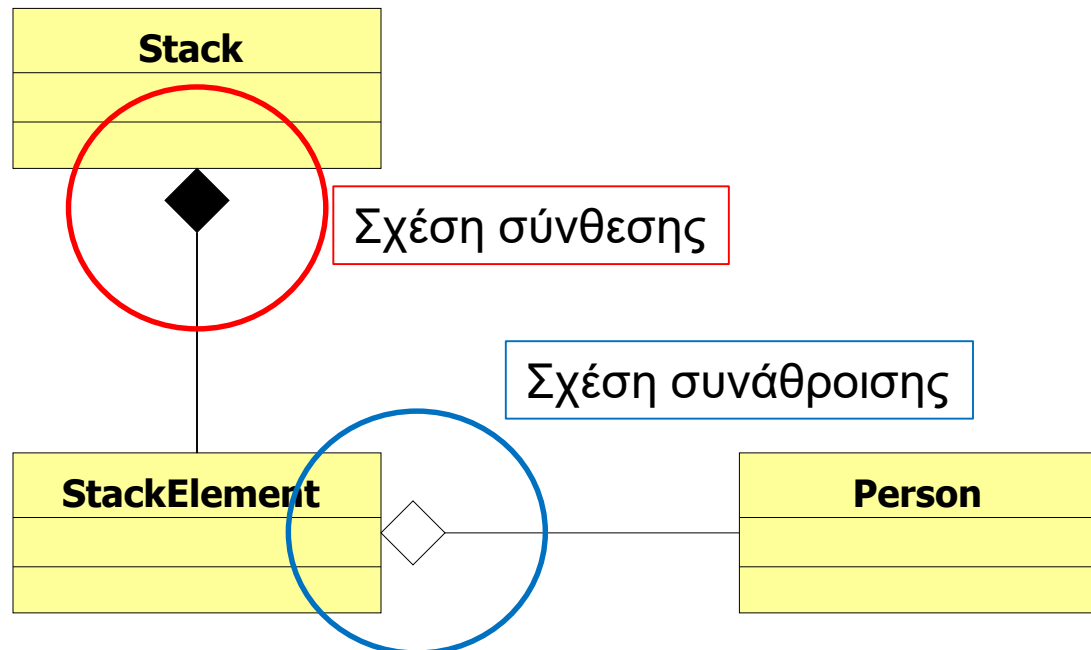
- Η κλάση **X** έχει σχέση **συνάθροισης** με την κλάση **Y**, αν αντικείμενο/α της κλάσης **Y** **ανήκουν στο** αντικείμενο της κλάσης **X**.
 - Τα αντικείμενα της κλάσης **Y** **έχουν υπόσταση και εκτός** της κλάσης **X**.
 - Όταν καταστρέφεται ένα αντικείμενο της κλάσης **X** **δεν καταστρέφονται απαραίτητα** και τα αντικείμενα της κλάσης **Y**.
- Παραδείγματα:
 - Σε έναν άνθρωπο μπορεί να ανήκει ένα αυτοκίνητο, ρούχα, κλπ.
 - Ένα κτήριο μπορεί να έχει μέσα ανθρώπους, έπιπλα, κλπ.
- Στην περίπτωση μας η κλάση **StackElement** έχει σχέση συνάθροισης με την κλάση **Person**.

Σχέση σύνθεσης – Composition

- Η κλάση **X** έχει σχέση σύνθεσης με την κλάση **Y**, αν το αντικείμενο της κλάσης **X** **αποτελείται από** αντικείμενα της κλάσης **Y**.
 - Τα αντικείμενα της κλάσης **Y** **δεν υπάρχουν εκτός** της κλάσης **X**.
 - Η κλάση **X** **δημιουργεί** τα αντικείμενα της κλάσης **Y**, και **καταστρέφονται** όταν καταστρέφεται το αντικείμενο της κλάσης **X**.
- Παραδείγματα:
 - Ένας άνθρωπος αποτελείται από μέρη του σώματος: κεφάλι, πόδια, χέρια κλπ.
 - Ένα κτήριο αποτελείται από τοίχους, δωμάτια, πόρτες, κλπ.
- Στην περίπτωση μας η κλάση **Stack** έχει σχέση σύνθεσης με την κλάση **StackElement**.

UML διαγράμματα

- Για να ξεχωρίζουν μεταξύ τους (κάποιες φορές) αναπαριστώνται διαφορετικά στα **UML** διαγράμματα.



Aggregation and Composition

- Το αν θα είναι μια σχέση, σχέση **συνάθροισης** ή **σύνθεσης** εξαρτάται κατά πολύ και από την υλοποίηση μας και τον σχεδιασμό.
 - Π.χ., σε ένα διαφορετικό πρόγραμμα μπορεί να επαναχρησιμοποιούμε το StackElement.
 - Π.χ., σε μία διαφορετική εφαρμογή, τα ανθρώπινα όργανα υπάρχουν και χωρίς τον άνθρωπο.

Προσοχή!

- Ο διαχωρισμός σε σχέσεις συνάθροισης και σύνθεσης είναι ως ένα βαθμό ένας **φορμαλισμός**.
 - Μην «κολλήσετε» προσπαθώντας να ορίσετε την σχέση.
 - Το σημαντικό είναι όταν δημιουργείτε το πρόγραμμα σας να σκεφτείτε **ποιες κλάσεις χρειάζονται τα αντικείμενα** που δημιουργούνται και **πότε πρέπει να δημιουργηθούν** μέσα στον κώδικα, και ποιες κλάσεις επηρεάζονται όταν αλλάζουν.
 - **Δεν υπάρχει χρυσός κανόνας**. Γενικά το πώς θα σχεδιαστεί το πρόγραμμα είναι κάτι που μπορεί να γίνει με πολλούς τρόπους συνήθως. Διαλέξτε αυτόν που θα κάνει το πρόγραμμα πιο **απλό**, **ευανάγνωστο**, **εύκολο να επεκταθεί**, να **ξαναχρησιμοποιηθεί** και να **διατηρηθεί**.

ΠΑΡΑΔΕΙΓΜΑ ΣΥΝΘΕΣΗΣ

Μεγάλο παράδειγμα

- Θέλουμε να δημιουργήσουμε ένα λογισμικό για ένα τμήμα πανεπιστημίου. Το τμήμα έχει 4 φοιτητές οπού ο καθένας έχει ένα όνομα και ένα αριθμό μητρώου (AM), και 2 καθηγητές που ο καθένας έχει ένα όνομα και ένα ΑΦΜ. Το τμήμα δίνει 2 μαθήματα. Το κάθε μάθημα έχει κωδικό και όνομα και κάποιες διδακτικές μονάδες. Το κάθε μάθημα ανατίθεται σε ένα καθηγητή. Οι φοιτητές γράφονται σε κάποιο μάθημα και αν περάσουν το μάθημα παίρνουν τις μονάδες. Θέλουμε να μπορούμε να τυπώσουμε τις πληροφορίες για το μάθημα: το όνομα, τον καθηγητή και τη λίστα των φοιτητών που παίρνουν το μάθημα.

Μεγάλο Παράδειγμα

- Θέλουμε να δημιουργήσουμε ένα λογισμικό για ένα **τμήμα** πανεπιστημίου.
- Το τμήμα έχει 4 **φοιτητές** όπου ο καθένας έχει ένα **όνομα** και ένα **αριθμό μητρώου** (AM).
- Το τμήμα έχει 2 **καθηγητές** που ο καθένας έχει ένα **όνομα** και ένα **ΑΦΜ**.
- Το τμήμα δίνει 2 **μαθήματα**. Το κάθε μάθημα έχει **κωδικό** και **όνομα**, και κάποιες **διδασκτικές μονάδες**.
- Το κάθε μάθημα **ανατίθεται** σε ένα καθηγητή.
- Οι φοιτητές **γράφονται** σε κάποιο μάθημα και αν **περάσουν** θα **πάρουν** τις μονάδες.
- Θέλουμε να μπορούμε να **τυπώσουμε** τις πληροφορίες του μαθήματος: το **όνομα**, τον **καθηγητή** και τη **λίστα** των **φοιτητών** που παίρνουν το μάθημα.

Κλάσεις μέθοδοι και πεδία

- Ουσιαστικά:
 - Τμήμα
 - Φοιτητές
 - Καθηγητές
 - Μαθήματα
 - Όνομα
 - ΑΜ, ΑΦΜ, κωδικός
 - Βαθμός
 - Λίστα φοιτητών
- Τα ουσιαστικά είναι υποψήφια για κλάσεις ή πεδία
- Ρήματα:
 - Ανατίθεται
 - Εγγράφεται
 - Τυπώνει
 - Περνάω μάθημα
 - Παίρνω μονάδες
- Τα ρήματα είναι υποψήφια για να γίνουν μέθοδοι και μηνύματα μεταξύ αντικειμένων.

Κλάσεις μέθοδοι και πεδία

- Ουσιαστικά:
 - Τμήμα
 - Φοιτητές
 - Καθηγητές
 - Μαθήματα
 - Όνομα
 - ΑΜ, ΑΦΜ, κωδικός
 - Βαθμός
 - Λίστα φοιτητών
 - Τα ουσιαστικά είναι υποψήφια για κλάσεις ή πεδία
- Ρήματα:
 - Ανατίθεται
 - Εγγράφεται
 - Τυπώνει
 - Περνάω μάθημα
 - Παίρνω μονάδες
 - Τα ρήματα είναι υποψήφια για να γίνουν μέθοδοι και μηνύματα μεταξύ αντικειμένων.

Όλα τα ουσιαστικά μπορούν να γίνουν κλάσεις αλλά συνήθως διαλέγουμε αυτά για τα οποία υπάρχει αρκετή πολυπλοκότητα

Κλάση Professor

- Κρατάει το όνομα και το ΑΦΜ του καθηγητή
- Ενδεχομένως να κρατάει και τα μαθήματα που έχει αναλάβει
- Η μέθοδος για να αναλάβει ο καθηγητής ένα μάθημα θα πρέπει να είναι εδώ ή στην κλάση του μαθήματος?

Κλάση Student

- Κρατάει το όνομα του φοιτητή και τις μονάδες που έχει πάρει μέχρι τώρα.
- Ενδεχομένως να κρατάει και τα μαθήματα που παίρνει.
- Ενδεχομένως να κρατάει και τη λίστα με τα μαθήματα που έχει περάσει.
- Χρειαζόμαστε μέθοδο για να γραφτεί ο φοιτητής στο μάθημα, ή να το περάσει, ή καλύτερα να τις βάλουμε στην κλάση του μαθήματος?

Κλάση Course

- Κρατάει το όνομα του μαθήματος, τις μονάδες του μαθήματος, τον καθηγητή που κάνει το μάθημα, τους φοιτητές που παίρνουν το μάθημα
 - Τίποτα άλλο? Τι θα κάνουμε με τους βαθμούς και το ποιος πέρασε το μάθημα?
- Μέθοδοι
 - Ανάθεση καθηγητή
 - Εγγραφή φοιτητή στο μάθημα
 - Ανάθεση βαθμών στους φοιτητές.

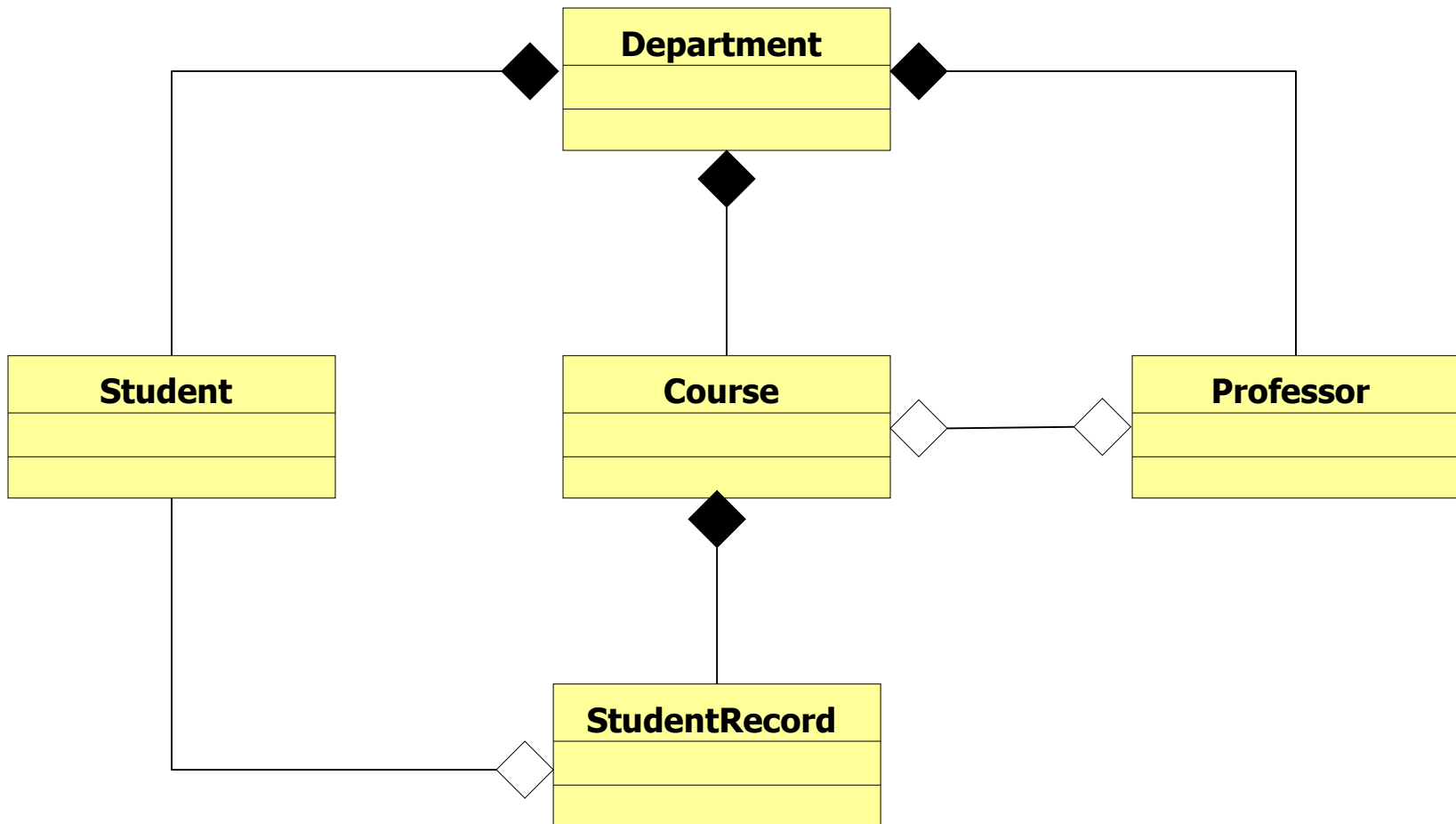
Κλάση Department

- Τα βάζει όλα μαζί, εδώ δημιουργούμε τους φοιτητές, καθηγητές, μαθήματα.
 - Οι φοιτητές και οι καθηγητές ως άτομα θα μπορούσαν να υπάρχουν και εκτός του τμήματος.
 - Εδώ δημιουργούμε την main.
-
- Χρειαζόμαστε άλλη κλάση?

Κλάση StudentRecord

- Χρειαζόμαστε να κρατάμε για κάθε φοιτητή τις πληροφορίες του (αυτά που έχουμε στο Student class) και το βαθμό του.
- Μας βολεύει να δημιουργήσουμε μια καινούρια κλάση που να βάζει μαζί αυτές τις πληροφορίες.

UML διάγραμμα



Αποθήκευση φοιτητών

- Η κλάση `Course` χρειάζεται να αποθηκεύσει τους φοιτητές που παίρνουν το μάθημα.
 - Δεν ξέρουμε εκ των προτέρων **πόσοι φοιτητές** θα πάρουν το μάθημα
- Θα χρησιμοποιήσουμε την κλάση `ArrayList` για να τους αποθηκεύσουμε.


```
public class Professor
{
    private String name;
    private int AFM;
    private Course lesson;

    public Professor(String name, int afm) {
        this.name = name;
        this.AFM = afm;
    }

    public void setLesson(Course c) {
        lesson = c;
    }

    public String toString() {
        return name + " " + AFM + " " + lesson;
    }
}
```

```
public class Student
{
    private String name;
    private int AM;
    private int units = 0;

    public Student(String name, int am){
        this.name = name;
        this.AM = am;
    }

    public int getAM(){
        return AM;
    }

    public void addUnits(int units){
        this.units += units;
    }

    public String toString(){
        return name + " AM:" + AM + " units:" + units;
    }
}
```

```
public class StudentRecord
{
    private Student student;
    private double grade;

    public StudentRecord(Student s){
        student = s;
    }

    public void setGrade(double grade){
        this.grade = grade;
    }

    public Student getStudent(){
        return student;
    }

    public String toString(){
        return student + " : " + grade;
    }

    public boolean passed(){
        if (grade >= 5){ return true;}
        return false;
    }
}
```

```
import java.util.ArrayList;
import java.util.Scanner;

public class Course
{
    private String name;
    private int code;
    private int units;
    private Professor prof;
    private ArrayList<StudentRecord> studentList
        = new ArrayList<StudentRecord>();

    public Course(String name, int code, int units){
        this.name = name;
        this.code = code;
        this.units = units;
    }

    public void setProf(Professor p)
    {
        prof = p;
        p.setLesson(this);
    }

    public void enroll(Student s){
        studentList.add(new StudentRecord(s));
    }
}
```

Χρησιμοποιούμε το this ως αναφορά στο παρόν αντικείμενο, ώστε να το προσθέσουμε στο αντικείμενο Professor

Δημιουργία του αντικειμένου StudentRecord και ταυτόχρονη προσθήκη στη λίστα. Λέγεται και «ανώνυμο αντικείμενο»

```

public void assignGrades(){
    System.out.println("Give grades for course "+toString());
    Scanner input = new Scanner(System.in);
    for(StudentRecord record: studentList){
        System.out.println("Give grade for student "
            + record.getStudent().getAM() +":");
        double grade = input.nextDouble();
        record.setGrade(grade);
        if (record.passed()){
            record.getStudent().addUnits(units);
        }
    }
}

```

Διασχίζουμε τη
λίστα των
φοιτητών

Αλυσιδωτές κλήσεις μεθόδων
Γίνεται εφόσον μια μέθοδος επιστρέφει αντικείμενο.

```

public String toString(){
    return name + " " + code + "("+units + ")";
}

public void printInfo(){
    System.out.println("Course " + name
        + " " + code + "("+units + ")");
    for (StudentRecord r: studentList){
        System.out.println(r);
    }
}
}

```

```
import java.util.Scanner;
```

```
class Department
```

```
{  
    public static void main(String[] args)
```

```
{  
    int numOfStudents = Integer.parseInt(args[0]);
```

```
    Professor profX = new Professor("Prof X", 2012);  
    Professor profY = new Professor("Prof Y", 2013);
```

```
    Course oop = new Course("oop", 212, 10);  
    Course intro = new Course("intro", 101, 5);
```

```
    Student[] students = new Student[numOfStudents];  
    Scanner input = new Scanner(System.in);  
    for (int i = 0; i < numOfStudents; i++){  
        System.out.print("Give student name: ");  
        String name = input.next();  
        students[i] = new Student(name, i);  
    }
```

```
    oop.setProf(profX);  
    oop.enroll(students[0]); oop.enroll(students[1]); oop.enroll(students[3]);
```

```
    intro.setProf(profY);  
    intro.enroll(students[2]); intro.enroll(students[3]);
```

```
    oop.assignGrades(); intro.assignGrades();
```

```
    System.out.println(profX); System.out.println(profY);  
    oop.printInfo(); intro.printInfo();
```

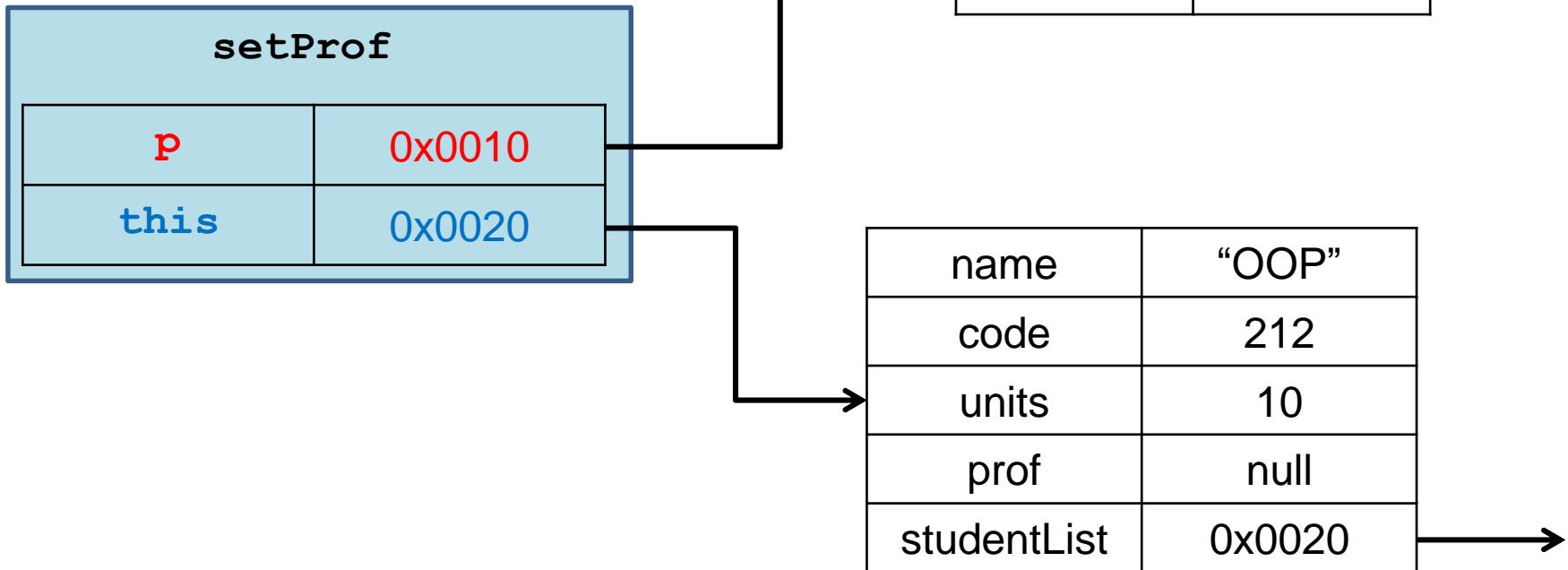
```
}
```

```
}
```

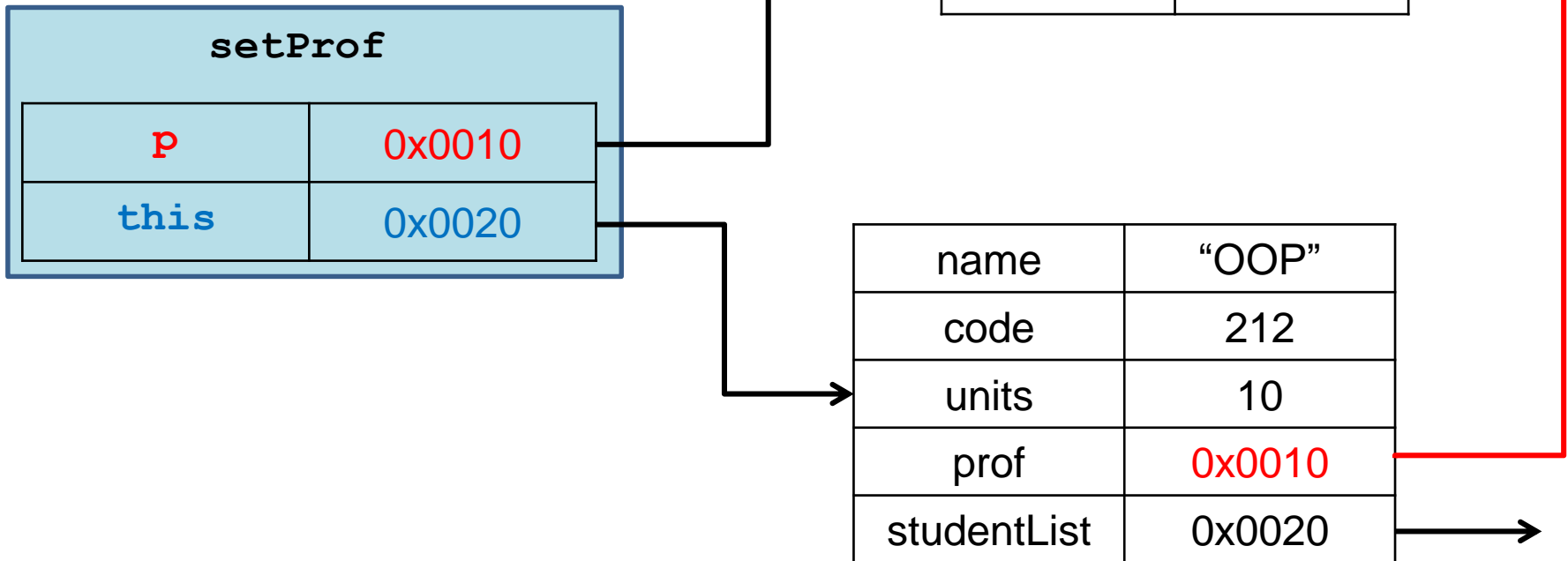
Χρησιμοποιούμε τις παραμέτρους εκτέλεσης (**command line arguments**) για να περάσουμε τον αριθμό των φοιτητών

Μετατρέπουμε το String σε ακέραιο με την μέθοδο **Integer.parseInt**

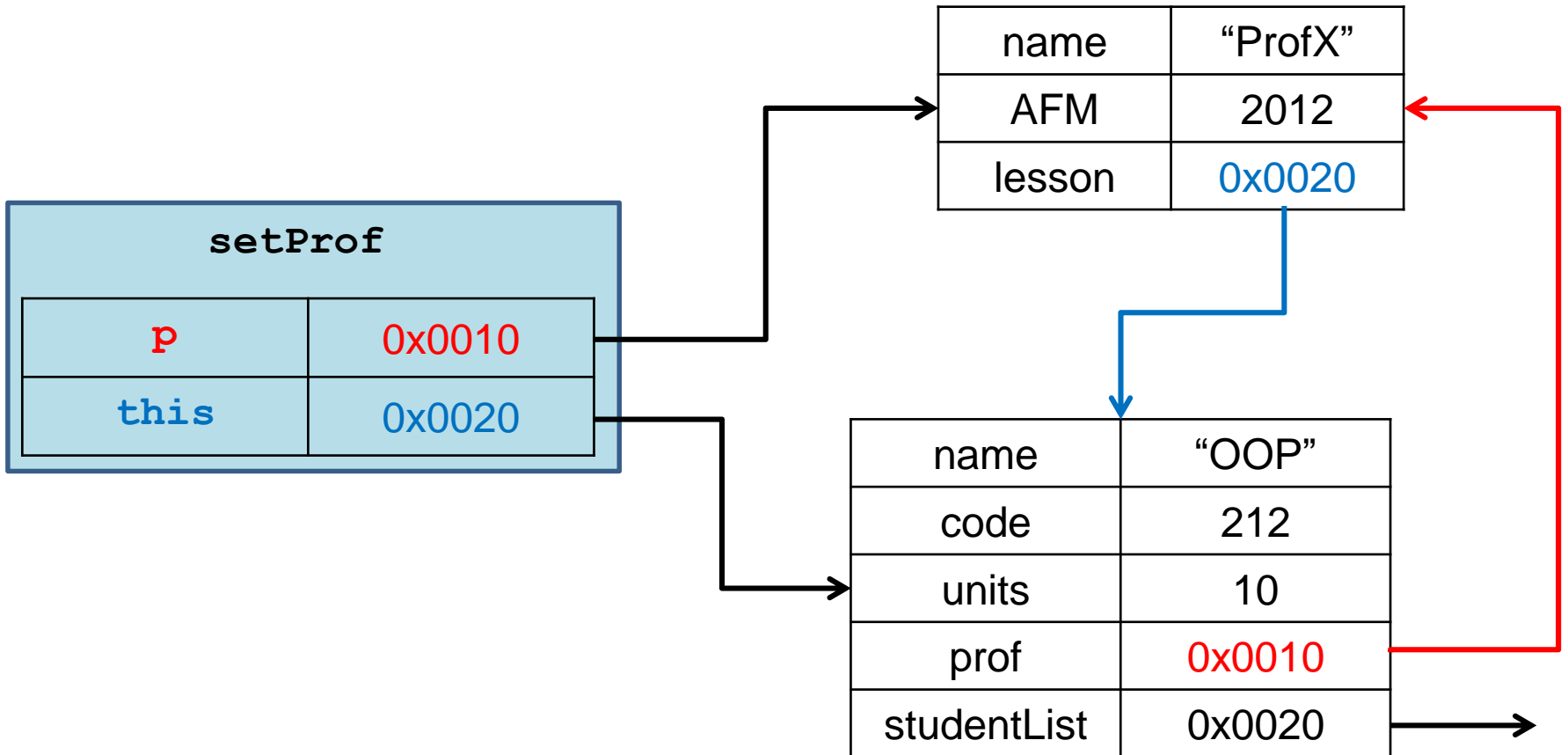
```
public void setProf(Professor p) {  
    prof = p;  
    p.setLesson(this);  
}
```



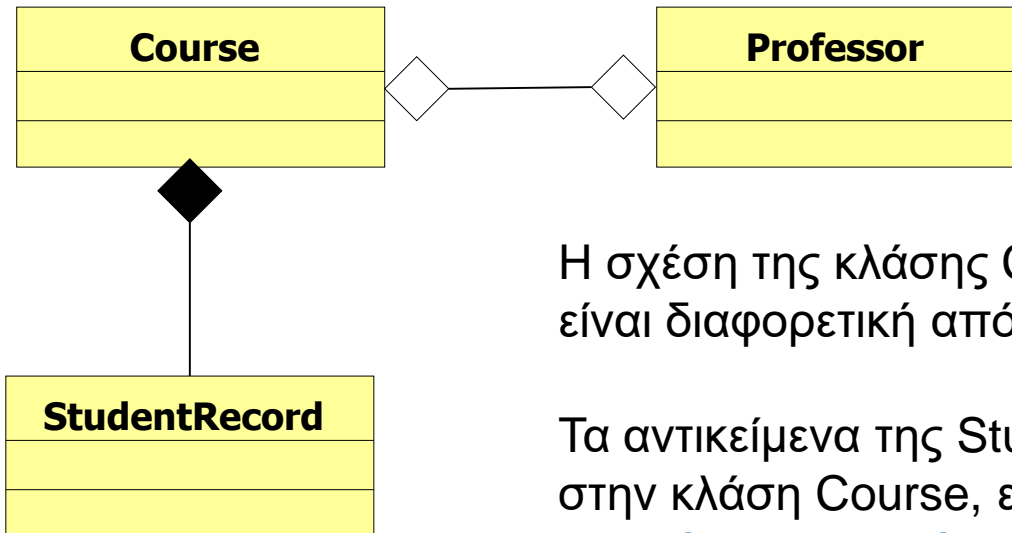
```
public void setProf(Professor p) {  
    prof = p;  
    p.setLesson(this);  
}
```




```
public void setProf(Professor p) {  
    prof = p;  
    p.setLesson(this);  
}
```



Σχέσεις κλάσεων



Η σχέση της κλάσης Course με την StudentRecord είναι διαφορετική από αυτή με την Professor

Τα αντικείμενα της StudentRecord **δημιουργούνται μέσα** στην κλάση Course, ενώ το αντικείμενο Professor **περνιέται ως παράμετρος** στην setProf

Προσοχή: Σε πολλά βιβλία και οι δύο σχέσεις αναφέρονται ως σχέση σύνθεσης!
Υπάρχει ποιοτική διαφορά παρότι το όνομα μπορεί να μην διαφέρει

Κάποιες φορές, η πρώτη σχέση λέγεται **σχέση σύνθεσης** και η δεύτερη **σχέση συνάθροισης**

Η σχέση Course και Professor είναι αμφίδρομη μιας και κρατάμε το αντικείμενο Course μέσα στην Professor

Αν θέλουμε ο φοιτητής να κρατάει
πληροφορία για το ποια μαθήματα παίρνει

```
public class Student
{
    private String name;
    private int AM;
    private int units = 0;
    ArrayList<Course> courses = new ArrayList<Course>();

    public Student(String name, int am){
        this.name = name;
        this.AM = am;
    }

    public String getName(){
        return name;
    }

    public void addUnits(int units){
        this.units += units;
    }

    public void addCourse(Course c){
        courses.add(c);
    }

    public String toString(){
        return name + " AM:" + AM + " units:" + units;
    }
}
```

```
import java.util.ArrayList;
import java.util.Scanner;

public class Course
{
    private String name;
    private int code;
    private int units;
    private Professor prof;
    private ArrayList<StudentRecord> studentList
        = new ArrayList<StudentRecord>();

    public Course(String name, int code, int units){
        this.name = name;
        this.code = code;
        this.units = units;
    }

    public void setProf(Professor p){
        prof = p;
        p.setLesson(this);
    }

    public void enroll(Student s){
        studentList.add(new StudentRecord(s));
        s.addCourse(this);
    }
}
```

Αναζήτηση

- Τι γίνεται αν θέλουμε να μπορούμε να ζητήσουμε τον βαθμό ενός φοιτητή για ένα μάθημα?
 - Η κλάση Course θα πρέπει να μπορεί με το AM του φοιτητή να μας επιστρέφει τον βαθμό.
 - Για τέτοιου είδους αναζητήσεις βολεύει να χρησιμοποιούμε ένα **λεξικό**.
 - Η Java μας προσφέρει την κλάση **HashMap** που υλοποιεί ένα λεξικό.

HashMap ([JavaDocs link](#))

- Αποθηκεύει ζευγάρια από κλειδιά και τιμές
- Constructors
 - `HashMap<K, V> myMap = new HashMap<K, V> ();`
- Μέθοδοι
 - `put(K key, V value)` : προσθέτει το ζευγάρι (`key, value`) (δημιουργεί μία συσχέτιση)
 - `V get(K key)` : επιστρέφει την τιμή για το κλειδί `key`.
 - `remove(K key)` : αφαιρεί το ζευγάρι με κλειδί `key`.
 - `containsKey(K key)` : boolean αν το σύνολο περιέχει το κλειδί `key` ή όχι.
 - `containsValue(V value)` : boolean αν το σύνολο περιέχει την τιμή `value` ή όχι. (αργό)
 - `size()` : ο αριθμός των στοιχείων (κλειδιών) στο map.
 - `isEmpty()` : boolean αν έχει στοιχεία το map ή όχι.
 - `Set<K> keySet()` : επιστρέφει ένα `Set` με τα κλειδιά.
 - `Collection<V> values()` : επιστρέφει ένα `Collection` με τις τιμές

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Scanner;
```

```
public class Course
{
```

```
    private String name;
    private int code;
    private int units;
    private Professor prof;
    private ArrayList<StudentRecord> studentList
        = new ArrayList<StudentRecord>();
    private HashMap<Integer, StudentRecord> studentMap
        = new HashMap<Integer, StudentRecord>();
```

```
    public void enroll(Student s){
        StudentRecord sRecord = new StudentRecord(s);
        studentList.add(sRecord);
        studentMap.put(s.getAM(), sRecord);
    }
```

```
    public double getGrade(int AM){
        if (studentMap.containsKey(AM)) {
            StudentRecord sRecord = studentMap.get(AM);
            return sRecord.getGrade();
        }else{
            System.out.println("Student "+ AM + " not enrolled");
            return -1;
        }
    }
}
```

Έχοντας το λεξικό μπορούμε να κάνουμε διάφορες αναζητήσεις με το AM του φοιτητή

Η ίδια αναφορά αποθηκεύεται σε δύο σημεία

8. ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ

Εισαγωγή στην Κληρονομικότητα

Is-a σχέση

Υπέρβαση μεθόδων

Downcasting

Παράδειγμα

- Στο παράδειγμα με το τμήμα πανεπιστημίου οι **φοιτητές** και οι **καθηγητές** είχαν κάποια **κοινά** στοιχεία
 - Και οι δύο είχαν όνομα
 - Και οι δύο είχαν κάποιο χαρακτηριστικό αριθμό
- και κάποιες **διαφορές**
 - Οι καθηγητές δίδασκαν μαθήματα
 - Οι φοιτητές έπαιρναν μαθήματα, βαθμούς και μονάδες
- Παρομοίως μπορούμε να έχουμε πολλούς διαφορετικούς τύπους φοιτητών ανάλογα με το έτος τους ή τις απαιτήσεις του μαθήματος
- Δεν θα ήταν βολικό αν είχαμε μεθόδους που να χειρίζονταν με **κοινό τρόπο τις ομοιότητες** (π.χ. εκτύπωση των βασικών στοιχείων) και να έχουν **ξεχωριστές μεθόδους για τις διαφορές**?
 - Έτσι δεν θα έπρεπε να γράφουμε τον **ίδιο κώδικα** πολλές φορές και οι **αλλαγές** θα έπρεπε να γίνουν μόνο μια φορά.
- Αυτό το καταφέρνουμε με την **κληρονομικότητα!**

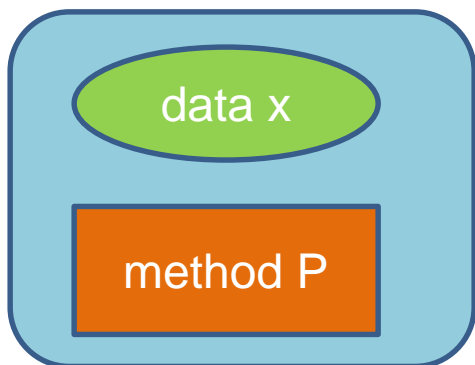
Κληρονομικότητα

- Η **κληρονομικότητα** είναι κεντρική έννοια στον αντικειμενοστραφή προγραμματισμό.
- Η ιδέα είναι να ορίσουμε μια **γενική κλάση** που έχει κάποια χαρακτηριστικά (πεδία και μεθόδους) που θέλουμε και μετά να ορίσουμε **εξειδικευμένες παραλλαγές** της κλάσης αυτής στις οποίες προσθέτουμε ειδικότερα χαρακτηριστικά.
 - Οι εξειδικευμένες κλάσεις λέμε ότι **κληρονομούν** τα χαρακτηριστικά της γενικής κλάσης

Κληρονομικότητα

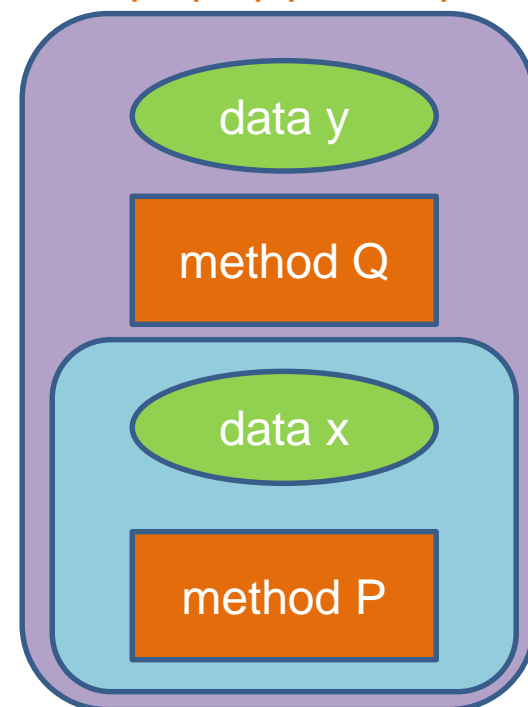
Έχουμε μια **Βασική Κλάση (Base Class) B**, με κάποια πεδία και μεθόδους.

Βασική Κλάση B



Θέλουμε να δημιουργήσουμε μια νέα κλάση D η οποία να έχει όλα τα χαρακτηριστικά της B, αλλά και κάποια επιπλέον.

Παράγωγη Κλάση D



Αντί να ξαναγράψουμε τον ίδιο κώδικα δημιουργούμε μια **Παράγωγη Κλάση (Derived Class) D**, η οποία **κληρονομεί** όλη τη λειτουργικότητα της Βασικής Κλάσης B και στην οποία προσθέτουμε τα νέα πεδία και μεθόδους.

Αυτή διαδικασία λέγεται **κληρονομικότητα**

Κληρονομικότητα

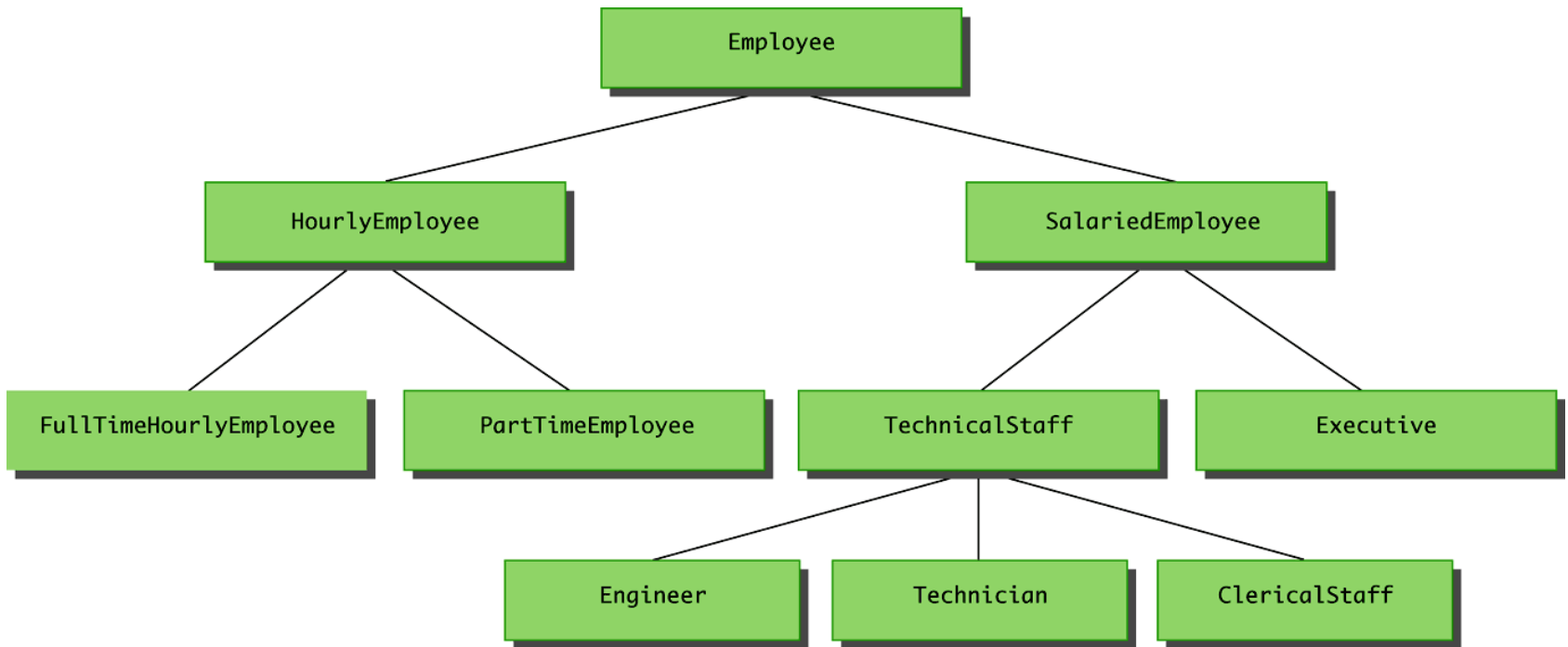
- Η κληρονομικότητα είναι χρήσιμη όταν
 - Θέλουμε να έχουμε αντικείμενα και της κλάσης B και της κλάσης D.
 - Θέλουμε να ορίσουμε πολλαπλές παράγωγες κλάσεις D1, D2, ... που η κάθε μία επεκτείνει την B με διαφορετικό τρόπο.
- Μπορούμε να ορίσουμε παράγωγες κλάσεις των παράγωγων κλάσεων.
 - Με αυτό τον τρόπο ορίζεται μια ιεραρχία κλάσεων.

Ιεραρχία κλάσεων (Class Hierarchy)

- Παράδειγμα: Έχουμε ένα πρόγραμμα που διαχειρίζεται τους **Εργαζόμενους** μιας εταιρίας.
 - Όλοι οι εργαζόμενοι έχουν κοινά χαρακτηριστικά το όνομα τους και το ΑΦΜ τους.
- Οι εργαζόμενοι χωρίζονται σε **Ωρομίσθιους** και **Έμμισθους**
 - Διαφορετικά χαρακτηριστικά θα κρατάμε όσον αφορά το μισθό για τον καθένα
- Οι **Ωρομίσθιοι** χωρίζονται σε **Πλήρους** και **Μερικής απασχόλησης**
- Οι **Έμμισθοι** χωρίζονται σε **Τεχνικό Προσωπικό** και **Διευθυντικό προσωπικό**
- Κ.ο.κ.....

A Class Hierarchy

Display 7.1 A Class Hierarchy

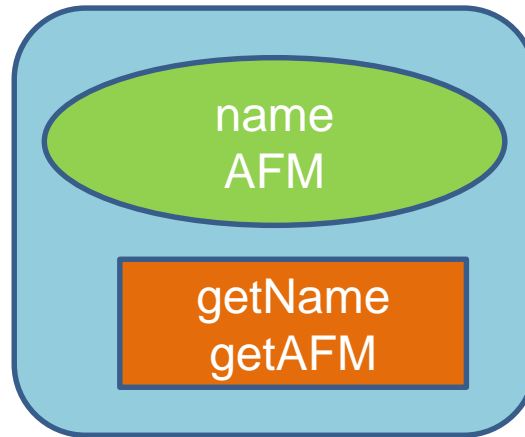


Ιεραρχία κλάσεων

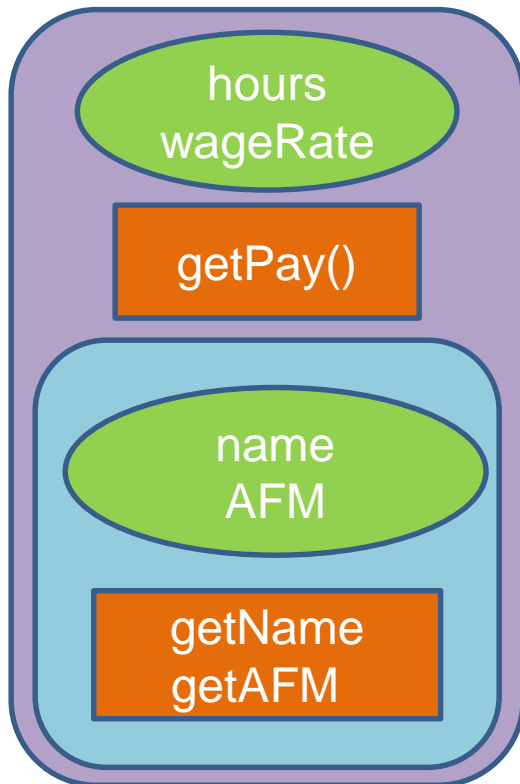
- Η **ιεραρχία** από κλάσεις ορίζει κάτι σαν **γενεαλογικό δέντρο κλάσεων** από πιο γενικές προς πιο ειδικές κλάσεις.
- Στη Java όλες οι κλάσεις ανήκουν στην ίδια ιεραρχία.
 - Στην κορυφή της ιεραρχίας είναι η κλάση **Object**.

Παράδειγμα

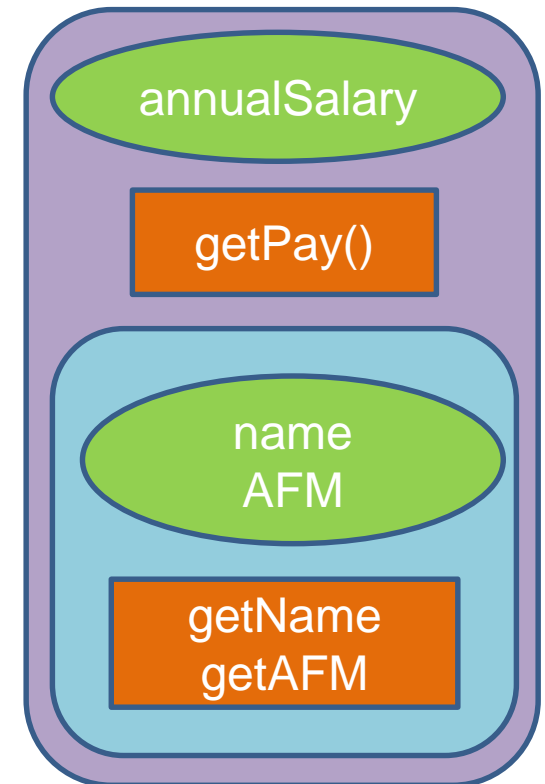
Employee



HourlyEmployee



SalariedEmployee



Οι παράγωγες κλάσεις κληρονομούν τα πεδία και τις μεθόδους της βασικής κλάσης

Πλεονέκτημα: επαναχρησιμοποίηση του κώδικα!

Ορολογία

- Η βασική κλάση συχνά λέγεται και **υπέρ-κλάση** (**superclass**) και η παραγόμενη κλάση **υπό-κλάση** (**subclass**).
- Επίσης η βασική κλάση λέμε ότι είναι ο **γονέας** της παραγόμενης κλάσης, και η παράγωγη κλάση το **παιδί** της βασικής.
 - Αν έχουμε παραπάνω από ένα επίπεδο κληρονομικότητας στην ιεραρχία, τότε έχουμε **πρόγονο** και **απόγονο** κλάση.

ΣΥΝΤΑΚΤΙΚΟ

- Ας πούμε ότι έχουμε την βασική κλάση **Employee** και τις παραγόμενες κλάσεις **HourlyEmployee** και **SalariedEmployee**.
- Για να ορίσουμε τις παραγόμενες κλάσεις χρησιμοποιούμε το εξής συντακτικό στη δήλωση της κλάσης:

- `public class HourlyEmployee extends Employee`
- `public class SalariedEmployee extends Employee`

Η βασική κλάση

```
public class Employee
{
    private String name;
    private int AFM;

    public Employee( ) { ... }

    public Employee(String theName, int theAFM) { ... }

    public Employee(Employee originalObject) { ... }

    public String getName( ) { ... }
    public void setName(String newName) { ... }

    public int getAFM( ) { ... }
    public void setAFM (int newAFM) { ... }

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

Η παράγωγη κλάση HourlyEmployee

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee( ) { ... }

    public HourlyEmployee(String theName, int theAFM,
        double theWageRate, double theHours) { ... }

    public HourlyEmployee(HourlyEmployee originalObject) { ... }

    public double getRate( ) { ... }
    public void setRate(double newWageRate) { ... }

    public double getHours( ) { ... }
    public void setHours(double hoursWorked) { ... }

    public double getPay( ) {
        return wageRate*hours;
    }

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

Νέα πεδία για την
HourlyEmployee

Μέθοδος getPay
υπολογίζει το μηνιαίο
μισθό

Η παράγωγη κλάση SalariedEmployee

```
public class SalariedEmployee extends Employee
```

```
{
```

```
    private double salary; //annual
```

```
    public SalariedEmployee( ) { ... }
```

```
    public SalariedEmployee(String theName,  
                             int theAFM, double theSalary) { ... }
```

```
    public SalariedEmployee(SalariedEmployee originalObject ) { ... }
```

```
    public double getSalary( ) { ... }
```

```
    public void setSalary(double newSalary) { ... }
```

```
    public double getPay( )
```

```
    {
```

```
        return salary/12;
```

```
    }
```

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

```
}
```

Νέα πεδία για την
SalariedEmployee

Μέθοδος getPay υπολογίζει
το μηνιαίο μισθό.
Διαφορετική από την
προηγούμενη

```
public class Example1
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
                                                    100, 50.5, 40);

        SalariedEmployee bob = new SalariedEmployee("Bob",
                                                    200, 100000);

        System.out.println("Alice: "
            + alice.getName() + " "
            + alice.getAFM() + " "
            + alice.getPay());

        System.out.println("Bob: "
            + bob.getName() + " "
            + bob.getAFM() + " "
            + bob.getPay());
    }
}
```

Μέθοδοι της Employee

Μέθοδοι των παράγωγων κλάσεων

Constructor

```
public class Employee
{
    private String name;
    private int AFM;

    public Employee()
    {
        name = "no name";
        AFM = 0;
    }

    public Employee(String theName, int theAFM)
    {
        if (theName == null || theAFM <= 0)
        {
            System.out.println("Fatal Error creating employee.");
            System.exit(0);
        }
        name = theName;
        AFM = theAFM;
    }
}
```

```

public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee(String theName, int theAFM,
                           double theWageRate, double theHours)
    {
        super(theName, theAFM);
        if ((theWageRate >= 0) && (theHours >= 0))
        {
            wageRate = theWageRate;
            hours = theHours;
        }
        else
        {
            System.out.println(
                "Fatal Error: creating an illegal hourly employee.");
            System.exit(0);
        }
    }
}

```

Με τη λέξη κλειδί **super** αναφερόμαστε στην βασική κλάση.

Εδώ καλούμε τον **constructor** της Employee με ορίσματα το όνομα και το ΑΦΜ

Ο constructor **super** μπορεί να κληθεί **μόνο στην αρχή** της μεθόδου.


```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName,
                             int theAFM, double theSalary)
    {
        super(theName, theAFM);
        if (theSalary >= 0)
            salary = theSalary;
        else
        {
            System.out.println(
                "Fatal Error: Negative salary.");
            System.exit(0);
        }
    }
}
```

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee()
    {
        super();
        salary = 0;
    }
}
```

Καλεί τον default constructor της Employee

Η εντολή δεν είναι απαραίτητη σε αυτή την περίπτωση. Αν δεν έχουμε κάποια κλήση προς τον constructor της γονικής κλάσης, τότε καλείται εξ ορισμού ο default constructor της Employee.

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName,int theAFM)
    {
        salary = 0;
    }
}
```

Πως θα αρχικοποιηθεί το αντικείμενο στην περίπτωση που κληθεί αυτός ο constructor?

Εφόσον δεν καλούμε εμείς κάποιο constructor της γονικής κλάσης θα κληθεί ο default constructor ο οποίος θα αρχικοποιήσει το όνομα στο "no name" και το ΑΦΜ στο μηδέν.

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName,int theAFM)
    {
        super(theName, theAFM);
        salary = 0;
    }
}
```

Αν θέλουμε να αρχικοποιήσουμε το όνομα και το ΑΦΜ θα πρέπει να καλέσουμε τον αντίστοιχο constructor της γονικής κλάσης.

Constructor this

- Όπως καλείται ο constructor **super** της γονικής κλάσης μπορούμε να καλέσουμε και τον constructor **this** της ίδιας κλάσης.

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName, int theAFM, double theSalary)
    {
        super(theName, theAFM);
        if (theSalary >= 0)
            salary = theSalary;
        else{
            System.out.println("Fatal Error: Negative salary.");
            System.exit(0);
        }
    }

    public SalariedEmployee(){
        this("no name", 0, 0);
    }
}
```

Καλεί ένα άλλο constructor της ίδιας κλάσης

Γιατί να μην κάνουμε κάτι πιο απλό? Κατευθείαν ανάθεση των πεδίων

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee(String theName,
                             int theAFM, double theSalary)
    {
        name = theName;
        AFM = theAFM;
        salary = theSalary;
    }
}
```

ΛΑΘΟΣ!

Οι παραγόμενες κλάσεις **δεν** έχουν πρόσβαση στα **private** πεδία και τις **private** μεθόδους της βασικής κλάσης.

Κληρονομικότητα και ενθυλάκωση

- Οι **παραγόμενες** κλάσεις κληρονομούν την **πληροφορία** που έχει και η **γονική** κλάση
 - Ένα αντικείμενο SalariedEmployee έχει πληροφορία για το όνομα και το ΑΦΜ του υπαλλήλου.
- **Δεν έχουν** όμως **πρόσβαση** να διαβάσουν και να αλλάξουν ότι είναι **private** μέσα στην γονική κλάση.
 - Στην περίπτωση του SalariedEmployee, δεν μπορούμε να αλλάξουμε ή να διαβάσουμε το όνομα. Θα πρέπει να χρησιμοποιήσουμε τις **public μεθόδους** setName, getName.
 - Για τον constructor πρέπει να καλέσουμε την super.
- Με αυτό τον τρόπο **προστατεύουμε** τα δεδομένα της γονικής κλάσης από κώδικα εκτός της κλάσης.
- Ο περιορισμός ισχύει και για **μεθόδους** που είναι **private** στην γονική κλάση.

```
public class Employee
{
    private void doSomething() {
        System.out.println("doSomething");
    }
}
```

```
public class SalariedEmployee extends Employee
{
    public void doSomethingMore() {
        doSomething();
        System.out.println("and more");
    }
}
```

ΛΑΘΟΣ!

Protected μέλη

- Οι παράγωγες κλάσεις έχουν **πρόσβαση** σε όλα τα **public** πεδία και μεθόδους της γενικής κλάσης.
- **ΔΕΝ** έχουν πρόσβαση στα **private** πεδία και μεθόδους.
 - Μόνο μέσω public μεθόδων **set*** και **get***
- **Protected**: αν κάποια **πεδία** και **μέθοδοι** είναι protected μπορούν να τα δουν όλοι οι **απόγονοι** της κλάσης.
 - Το βιβλίο δεν το συνιστά.
- **Package access**: αν δεν προσδιορίσετε public, private, ή protected access τότε η default συμπεριφορά είναι ότι η μεταβλητή είναι προσβάσιμη από άλλες κλάσεις **μέσα στο ίδιο πακέτο**.

Employee

```
public class Employee
{
    private String name = "default";
    private Date hireDate = new Date(11,4,2013);

    public Employee()
    {
        System.out.println("empty constructor");
    }

    public Employee(String theName, Date theDate)
    {
        name = theName;
        hireDate = new Date(theDate);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee(String theName, Date theDate,
                           double theWageRate, double theHours)
    {
        name = theName;
        hireDate = new Date(theDate);
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

Χτυπάει λάθος η πρόσβαση σε **private** πεδία.

Employee

```
public class Employee
{
    protected String name = "default";
    protected Date hireDate = new Date(11,4,2013);

    public Employee()
    {
        System.out.println("empty constructor");
    }

    public Employee(String theName, Date theDate)
    {
        name = theName;
        hireDate = new Date(theDate);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee(String theName, Date theDate,
                           double theWageRate, double theHours)
    {
        name = theName;
        Date = new (theDate);
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

OK η πρόσβαση σε **protected** πεδία.

Πολλαπλοί τύποι

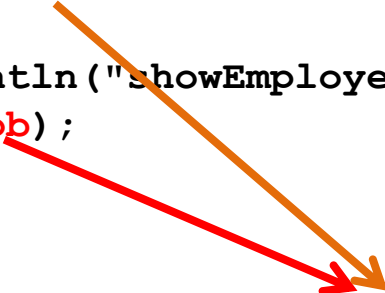
- Ένα αντικείμενο της παράγωγης κλάσης έχει **και** τον τύπο της βασικής κλάσης
 - Ένας HourlyEmployee είναι **και** Employee
 - Υπάρχει μία **is-a** σχέση μεταξύ των κλάσεων.
- Αυτό μπορούμε να το εκμεταλλευτούμε χρησιμοποιώντας την **βασική κλάση** όταν θέλουμε να χρησιμοποιήσουμε **κάποια** από τις **παράγωγες**.

```
public class IsADemo
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
                                                    100, 50.5, 40);
        SalariedEmployee bob = new SalariedEmployee("Bob",
                                                    200, 100000);

        System.out.println("showEmployee (alice) :");
        showEmployee (alice);

        System.out.println("showEmployee (bob) :");
        showEmployee (bob);
    }

    public static void showEmployee (Employee employeeObject)
    {
        System.out.println(employeeObject.getName ( ));
        System.out.println(employeeObject.getAFM ( ));
    }
}
```



Μπορούμε να καλέσουμε τη μέθοδο και με **HourlyEmployee** και με **SalariedEmployee** γιατί και οι δύο είναι και **Employee**.

```
public class Employee
{
    private String name;
    private int AFM;

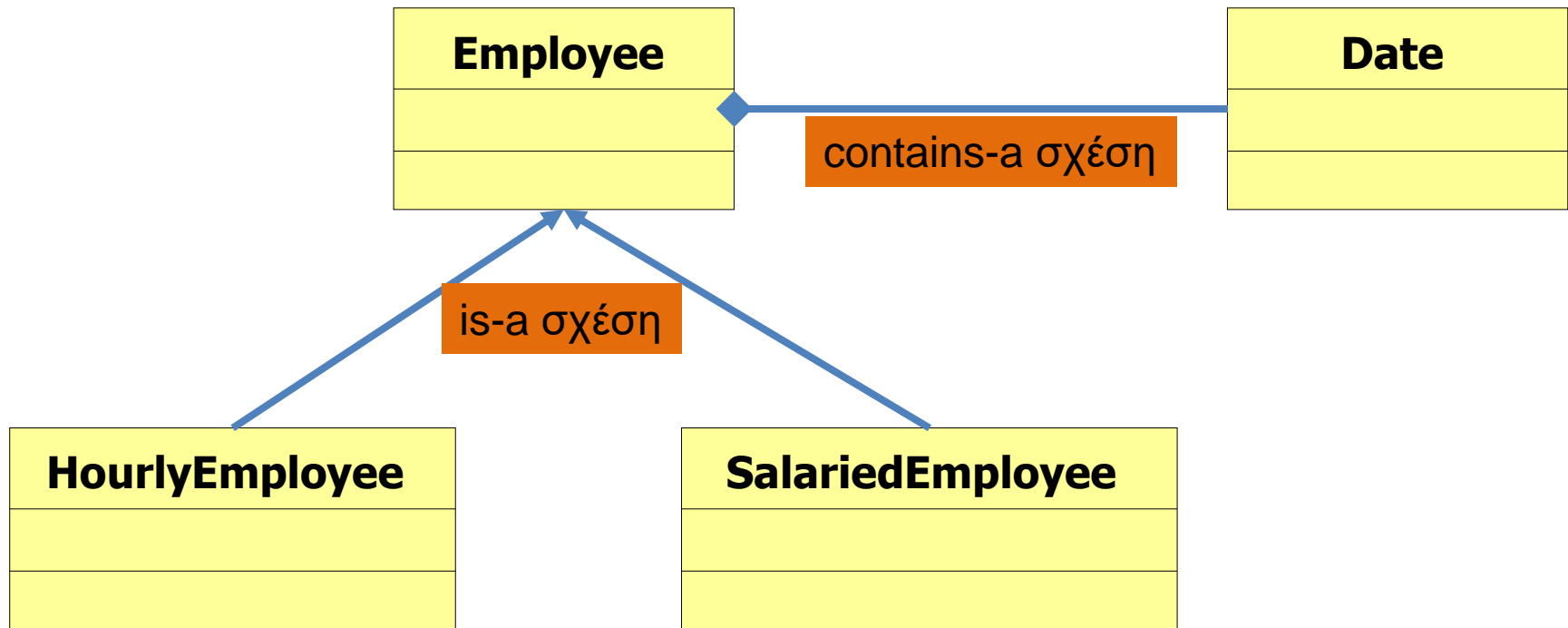
    public Employee(Employee other) {
        this.name = other.name;
        this.AFM = other.AFM;
    }
}
```

```
public class SalariedEmployee extends Employee
{
    public SalariedEmployee(SalariedEmployee other) {
        super(other);
        this.salay = other.salary;
    }
}
```

Η κλήση του copy constructor της Employee (μέσω της `super(other)`) γίνεται με ένα αντικείμενο τύπου SalariedEmployee. Αυτό γίνεται γιατί **SalariedEmployee is a Employee** και το αντικείμενο `other` έχει και τους δύο τύπους.

UML διάγραμμα

- Αναπαράσταση κληρονομικότητας



METHOD OVERRIDING

ΥΠΕΡΒΑΣΗ ΜΕΘΟΔΩΝ

Υπέρβαση μεθόδων (method overriding)

- Μία μέθοδος που ορίζεται στην βασική κλάση μπορούμε να την **ξανα-ορίσουμε** στην παράγωγη κλάση με διαφορετικό τρόπο
 - Παράδειγμα: η μέθοδος **toString()** . Την ξανα-ορίζουμε για κάθε παραγόμενη κλάση ώστε να παράγει αυτό που θέλουμε
 - Αυτό λέγεται **υπέρβαση** της μεθόδου (**method overriding**).
- Η **υπέρβαση** των μεθόδων είναι διαφορετική από την **υπερφόρτωση**.
 - Στην υπερφόρτωση **αλλάζουμε την υπογραφή** της μεθόδου.
 - Εδώ έχουμε την ίδια υπογραφή, απλά **αλλάζει ο ορισμός** στην παραγόμενη κλάση.

```
public class Employee
{
    private String name;
    private int AFM;

    public Employee( ) { ... }

    public Employee(String theName, int theAFM) { ... }

    public Employee(Employee originalObject) { ... }

    public String getName( ) { ... }
    public void setName(String newName) { ... }

    public Date getAFM( ) { ... }
    public void setAFM(int AFM) { ... }

    public String toString()
    {
        return (name + " " + AFM);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee( ) { ... }

    public HourlyEmployee(String theName, int theAFM,
        double theWageRate, double theHours) { ... }

    public HourlyEmployee(HourlyEmployee originalObject) { ... }

    public double getRate( ) { ... }
    public void setRate(double newWageRate) { ... }

    public double getHours( ) { ... }
    public void setHours(double hoursWorked) { ... }

    public double getPay( ) {
        return wageRate*hours;
    }

    public String toString( ){
        return (getName( ) + " " + getAFM( )
            + "\n$" + wageRate + " per hour for " + hours + " hours");
    }
}
```

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { ... }

    public SalariedEmployee(String theName,
                             int theAFM, double theSalary) { ... }

    public SalariedEmployee(SalariedEmployee originalObject ) { ... }

    public double getSalary( ) { ... }
    public void setSalary(double newSalary) { ... }

    public double getPay( )
    {
        return salary/12;
    }

    public String toString( ) { ... }
    {
        return (getName( ) + " " + getAFM( )
                + "\n$" + salary + " per year");
    }
}
```

```

public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { ... }

    public SalariedEmployee(String theName,
                               int theAFM, double theSalary) { ... }

    public SalariedEmployee(SalariedEmployee originalObject ) { ... }

    public double getSalary( ) { ... }
    public void setSalary(double newSalary) { ... }

    public double getPay( )
    {
        return salary/12;
    }

    public String toString( ) { ... }
    {
        return (super.toString( ) + "\n$" + salary + " per year");
    }
}

```

Έτσι καλούμε την toString της βασικής κλάσης
 Πιο καλή υλοποίηση, μπορεί να έχει φωλιασμένες
 κλήσεις από προγονικές κλάσεις

super

- Το keyword **super** χρησιμοποιείται σαν αντικείμενο κλήσης για να καλέσουμε μια μέθοδο της γονικής κλάσης την οποία έχουμε κάνει override.
 - Π.χ., `super.toString()` για να καλέσουμε την `toString` της `Employee`.
- Αν θέλουμε να το ξεχωρίσουμε από την κλήση της `toString` της `SalariedEmployee`, μπορούμε να χρησιμοποιήσουμε το **this**. Μέσα στην `SalariedEmployee`:
 - `super.toString()` καλεί την `toString` της `Employee`
 - `this.toString()` καλεί την `toString` της `SalariedEmployee`
- **Προσοχή: Δεν** μπορούμε να έχουμε **αλυσιδωτές** κλήσεις του `super`.
 - `super.super.toString()` είναι **λάθος!**

Παράδειγμα

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
        SalariedEmployee sam = new SalariedEmployee("Sam",
                                                    100, 100000);
        HourlyEmployee han = new HourlyEmployee("Han",
                                                200, 50.5, 40);
        Employee eve = new Employee("Eve", 300);

        System.out.println(eve);
        System.out.println(sam);
        System.out.println(han);
    }
}
```

Καλεί τη μέθοδο της Employee

Καλεί τη μέθοδο της SalariedEmployee

Καλεί τη μέθοδο της HourlyEmployee

Υπέρβαση και αλλαγή επιστρεφόμενου τύπου

- Μια αλλαγή που μπορούμε να κάνουμε στην υπογραφή της κλάσης που υπερβαίνουμε είναι να αλλάξουμε τον **επιστρεφόμενο τύπο** σε αυτόν μιας παράγωγης κλάσης
 - Ουσιαστικά δεν είναι αλλαγή αφού η παράγωγη κλάση έχει και τον τύπο της γονικής κλάσης.

```
public class Employee
{
    private String name;
    private Date hireDate;

    public Employee createCopy()
    {
        return new Employee(this);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee createCopy()
    {
        return new HourlyEmployee(this);
    }
}
```

Ο επιστρεφόμενος τύπος αλλάζει από **Employee** σε **HourlyEmployee** στην υπέρβαση. Ουσιαστικά όμως δεν υπάρχει αλλαγή μιας και κάθε αντικείμενο **HourlyEmployee** είναι και **Employee**

```
public class SalariedEmployee extends
Employee
{
    private double salary; //annual

    public SalariedEmployee createCopy()
    {
        return new SalariedEmployee(this);
    }
}
```

Ο επιστρεφόμενος τύπος αλλάζει από **Employee** σε **SalariedEmployee** στην υπέρβαση. Ουσιαστικά όμως δεν υπάρχει αλλαγή μιας και κάθε αντικείμενο **SalariedEmployee** είναι και **Employee**

toString και equals

- Είπαμε ότι η Java για κάθε αντικείμενο «περιμένει» να δει τις μεθόδους `toString` και `equals`
 - Αυτό σημαίνει ότι οι μέθοδοι αυτές ορίζονται στην κλάση `Object` που είναι ο πρόγονος όλων το κλάσεων και κάθε νέα κλάση μπορεί να τις **υπερβεί** (`override`).
 - Είδαμε παραδείγματα πως υπερβήκαμε την μέθοδο `toString`.

equals

- Η equals στην κλάση Object ορίζεται ως:
 - `public boolean equals(Object other)`
- Για την κλάση Employee θα την ορίσουμε ως:
 - `public boolean equals(Employee other)`
- Αλλάζουμε την **υπογραφή** της κλάσης, άρα δεν κάνουμε **υπέρβαση**, αλλά **υπερφόρτωση** της equals
 - Πως θα την ορίσουμε ώστε να κάνουμε υπέρβαση?

Overriding equals

```
public class Employee
{
    private String name;
    private Date hireDate;

    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass( ) != otherObject.getClass( ))
            return false;
        else
        {
            Employee otherEmployee = (Employee) otherObject;
            return (name.equals(otherEmployee.name)
                && hireDate.equals(otherEmployee.hireDate));
        }
    }
}
```

getClass: μέθοδος της Object, επιστρέφει μια αναπαράσταση της κλάσης του αντικειμένου

Downcasting: μετατροπή ενός αντικειμένου από μια υψηλότερη σε μία χαμηλότερη κλάση

Το downcasting δεν είναι πάντα δυνατόν και αν δεν γίνει σωστά μπορεί να προκαλέσει λάθη κατά την εκτέλεση του προγράμματος

DOWNCASTING

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
        SalariedEmployee sam = new SalariedEmployee("Sam",
            new Date(1, 1, 2010), 100000);
        Employee eve = new Employee("Eve", new Date(1,1,2012));

        SalariedEmployee eve2 = eve;
        if (sam.getHireDate().equals(eve2.getHireDate())){
            System.out.println("Same hire date");
        }else{
            System.out.println("Different hire date");
        }
    }
}
```

Στην περίπτωση αυτή προσπαθούμε να κάνουμε το downcasting έμμεσα, αναθέτοντας μια μεταβλητή Employee σε μια μεταβλητή SalariedEmployee. Θα μας χτυπήσει λάθος κατά την μεταγλώτιση.

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
        SalariedEmployee sam = new SalariedEmployee("Sam",
            new Date(1, 1, 2010), 100000);
        Employee eve = new Employee("Eve", new Date(1,1,2012));

        SalariedEmployee eve2 = (SalariedEmployee)eve;
        if (sam.getHireDate().equals(eve2.getHireDate())){
            System.out.println("Same hire date");
        }else{
            System.out.println("Different hire date");
        }
    }
}
```

Στην περίπτωση αυτή θα μας χτυπήσει λάθος στο τρέξιμο παρότι χρησιμοποιούμε μόνο την κοινή μέθοδο `getHireDate()`. Το πρόγραμμα προβλέπει ότι μπορεί να υπάρχει πρόβλημα. Δεν γίνεται να μετατρέψουμε έναν `Employee` σε `SalariedEmployee` (ο `Employee` δεν έχει όλα τα πεδία που χρειάζεται ένας `SalariedEmployee`)

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
        SalariedEmployee sam = new SalariedEmployee("Sam",
            new Date(1, 1, 2010), 100000);
        Employee eve = new Employee("Eve", new Date(1,1,2012));

        method(sam, sam);

    }

    private static void method(SalariedEmployee sEmp, Employee emp) {
        SalariedEmployee sEmp2 = (SalariedEmployee) emp;

        if (sEmp.getHireDate().equals(sEmp2.getSalary())) {
            System.out.println("Same Salary");
        }else{
            System.out.println("Different salary");
        }
    }
}
```

Στην περίπτωση αυτή το downcasting δεν χτυπάει λάθος γιατί **υπάρχει η δυνατότητα** να καλέσουμε σωστά την μέθοδο με SalariedEmployee αντικείμενο

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
        SalariedEmployee sam = new SalariedEmployee("Sam",
            new Date(1, 1, 2010), 100000);
        Employee eve = new Employee("Eve", new Date(1,1,2012));

        method(sam, eve);

    }

    private static void method(SalariedEmployee sEmp, Employee emp){
        SalariedEmployee sEmp2 = (SalariedEmployee) emp;

        if (sEmp.getHireDate().equals(sEmp2.getSalary())){
            System.out.println("Same Salary");
        }else{
            System.out.println("Different salary");
        }
    }
}
```

Αν όμως την καλέσουμε με αντικείμενο Employee θα πάρουμε λάθος

```
import java.util.Random;
```

```
public class DowncastingExample2
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        SalariedEmployee[] sEmployees = new SalariedEmployee[4];
```

```
        sEmployees[0] = new SalariedEmployee("employee 100", new Date(1, 1, 2015), 1000);
```

```
        sEmployees[1] = new SalariedEmployee("employee 101", new Date(2, 1, 2015), 2000);
```

```
        sEmployees[2] = new SalariedEmployee("employee 102", new Date(3, 1, 2015), 3000);
```

```
        sEmployees[3] = new SalariedEmployee("employee 103", new Date(4, 1, 2015), 4000);
```

```
        SalariedEmployee rand = (SalariedEmployee) randomSelection(sEmployees);
```

```
        System.out.println(rand);
```

```
        System.out.println("Salary per month " + rand.getPay());
```

```
    }
```

Σε τι μας χρειάζεται το downcasting?

Θέλουμε να καλέσουμε την μέθοδο `getPay` για τυπώσουμε τον μηνιαίο μισθό. Χρειαζόμαστε downcasting

```
private static Employee randomSelection(Employee[] employees) {
```

```
    Random rndGen = new Random();
```

```
    int r = rndGen.nextInt(employees.length);
```

```
    return employees[r];
```

```
}
```

```
}
```

Έχουμε μια γενική μέθοδο `randomSelection` που επιλέγει ένα τυχαίο στοιχείο από ένα πίνακα με `Employee`. Θέλουμε να την χρησιμοποιήσουμε σε ένα πίνακα με `SalariedEmployee`

Upcasting

- Η ανάθεση στην αντίθετη κατεύθυνση (**upcasting**) μπορεί να γίνει χωρίς να χρειάζεται casting
 - Μπορούμε να κάνουμε μια ανάθεση $x = y$ δύο αντικειμένων αν:
 - τα δύο αντικείμενα να είναι της **ίδιας κλάσης** ή
 - η κλάση του αντικειμένου που **ανατίθεται** (y) είναι **απόγονος** της κλάσης του αντικειμένου στο οποίο γίνεται η ανάθεση (x)
- Για παράδειγμα, ο παρακάτω κώδικας δουλεύει χωρίς πρόβλημα:
 - `Employee anEmployee;`
 - `Hourly Employee hEmployee = new HourlyEmployee();`
 - `anEmployee = hEmployee;`

```
public class IsADemo
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
                                                    100, 50.5, 40);
        SalariedEmployee bob = new SalariedEmployee("Bob",
                                                    200, 100000);

        System.out.println("showEmployee (alice) :");
        showEmployee (alice);

        System.out.println("showEmployee (bob) :");
        showEmployee (bob);
    }

    public static void showEmployee (Employee employeeObject)
    {
        System.out.println(employeeObject.getName ( ));
        System.out.println(employeeObject.getAFM ( ));
    }
}
```

Όταν καλούμε την `showEmployee` έμμεσα κάνουμε τις αναθέσεις:

```
employeeObject = alice
employeeObject = bob
```


9. ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ

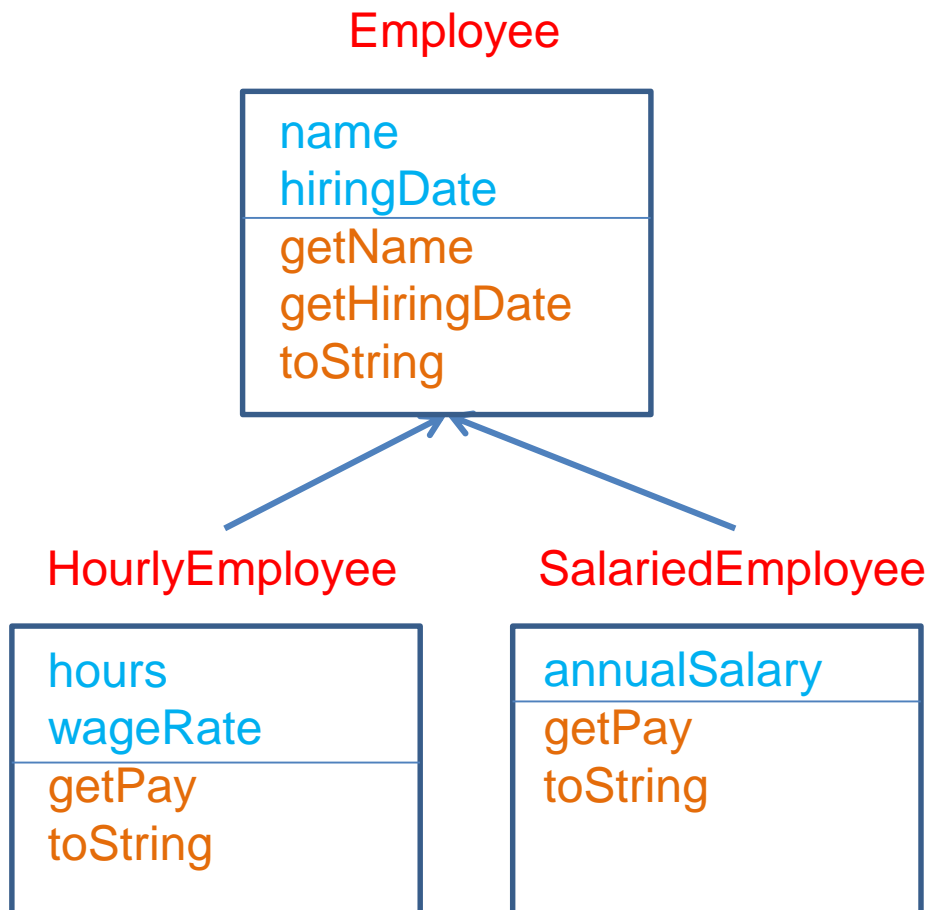
II

Πολυμορφισμός – Late Binding

Αφηρημένες κλάσεις

Interfaces – διεπαφές

Κληρονομικότητα



Οι παράγωγες κλάσεις κληρονομούν τα πεδία και τις μεθόδους της βασικής κλάσης και έχουν και δικά τους πεδία και μεθόδους.

Επίσης μπορούμε να υπερβαίνουμε (override) κάποιες μεθόδους (`toString`)

```
public class Employee
{
    private String name;
    private Date hireDate;

    public String toString(){
        return (name + " " + hireDate.toString( ));
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public String toString( ){
        return (super.toString( ) + "\n$" + wageRate + " per hour for " + hours + " hours");
    }
}
```

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public String toString( ){
        return (super.toString( ) + "\n$" + salary + " per year");
    }
}
```

ΠΟΛΥΜΟΡΦΙΣΜΟΣ LATE BINDING

```
public class IsADemo
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
                                                    100, 50.5, 40);
        SalariedEmployee bob = new SalariedEmployee("Bob",
                                                    200, 100000);

        System.out.println("showEmployee(alice):");
        showEmployee(alice);

        System.out.println("showEmployee(bob):");
        showEmployee(bob);
    }

    public static void showEmployee(Employee employeeObject)
    {
        System.out.println(employeeObject);
    }
}
```

Τι θα τυπώσει η `showEmployee` όταν την καλέσουμε με ορίσματα το `alice` και το `bob`?
Ποια μέθοδος `toString` θα κληθεί?

```

public class IsADemo
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
                                                    100, 50.5, 40);
        SalariedEmployee bob = new SalariedEmployee("Bob",
                                                    200, 100000);

        System.out.println("showEmployee (alice) :");
        showEmployee (alice);

        System.out.println("showEmployee (bob) :");
        showEmployee (bob);

    }

    public static void showEmployee(Employee employeeObject)
    {
        System.out.println(employeeObject);
    }
}

```

Θα καλέσει την `toString` της κλάσης του αντικειμένου που περνάμε σαν όρισμα (**HourlyEmployee** ή **SalariedEmployee**) και όχι την κλάση που εμφανίζεται στον ορισμό της παραμέτρου (**Employee**).

Ο μηχανισμός αυτός ονομάζεται **late binding** (και/ή **πολυμορφισμός**)

Late Binding (καθυστερημένη δέσμευση)

- Η **δέσμευση (binding)** αναφέρεται στον συσχετισμό μεταξύ της **κλήσης μιας μεθόδου** και του **ορισμού (κώδικα) της μεθόδου**.
- **Early binding:** Η δέσμευση γίνεται **κατά τη μεταγλώττιση** του προγράμματος
 - Στην περίπτωση αυτή η μέθοδος **toString()** που θα κληθεί θα είναι η μέθοδος της κλάσης **Employee** μιας και όταν γίνεται η μεταγλώττιση ο compiler βλέπει το όρισμα ως αντικείμενο της κλάσης **Employee**.
- **Late binding:** Η δέσμευση γίνεται **κατά τη εκτέλεση** του προγράμματος
 - Το κάθε αντικείμενο έχει **πληροφορία** για την κλάση του και τον ορισμό (κώδικα) των μεθόδων του.
 - Στην περίπτωση αυτή η μέθοδος **toString()** που θα κληθεί εξαρτάται από την κλάση που περνάμε σαν όρισμα (**Employee**, **HourlyEmployee** ή **SalariedEmployee**). Ανάλογα με το αντικείμενο καλείται η ανάλογη μέθοδος.
- Στη **Java** εφαρμόζεται ο μηχανισμός του **late binding για όλες τις μεθόδους** (σε αντίθεση με άλλες γλώσσες προγραμματισμού).

Παράδειγμα

```
public class Example3
{
    public static void main(String[] args)
    {
        Employee employeeArray[] = new Employee[3];

        employeeArray[0] = new Employee("alice",
                                         new Date(1,1,2010));

        employeeArray[1] = new HourlyEmployee("bob",
                                              new Date(1,1,2011), 20, 160);

        employeeArray[2] = new SalariedEmployee("charlie",
                                                new Date(1,1,2012), 24000);

        for (int i = 0; i < 3; i++){
            System.out.println(employeeArray[i]);
        }
    }
}
```

Για κάθε στοιχείο του πίνακα καλείται **διαφορετική** μέθοδος toString ανάλογα με το αντικείμενο που τοποθετήσαμε σε εκείνη τη θέση


```
public class mySale
{
    protected String name;
    protected double price;

    public mySale(String theName, double thePrice){
        name = theName;
        price = thePrice;
    }

    public String toString( ){
        return (name + " Price and total cost = $" + price);
    }

    public double bill( ){
        return price;
    }

    public boolean equalDeals(mySale otherSale){
        return (name.equals(otherSale.name)
            && this.bill( ) == otherSale.bill( ));
    }

    public boolean lessThan (mySale otherSale){
        return (this.bill( ) < otherSale.bill( ));
    }
}
```

Σύμφωνα με το βιβλίο δεν συνίσταται η χρήση της `protected` αλλά την χρησιμοποιούμε για απλότητα στο παράδειγμα

```
public class myDiscountSale extends mySale
{
    private double discount;

    public myDiscountSale(String theName,
                           double thePrice, double theDiscount)
    {
        super(theName, thePrice);
        discount = theDiscount;
    }
}
```

```
public double bill( )
{
    double fraction = discount/100;
    return (1 - fraction)*price;
}
```

Υπέρβαση της μεθόδου **bill()**

```
public String toString( )
{
    return (name + " Price = $" + price
           + " Discount = " + discount + "%\n"
           + " Total cost = $" + bill( ));
}
```

Δεν έχουμε υπέρβαση των μεθόδων **equalDeals** και **lessThan**

```

public class myLateBindingDemo
{
    public static void main(String[] args)
    {
        mySale simple = new mySale("floor mat", 10.00); //One item at $10.00.
        myDiscountSale discount = new myDiscountSale("floor mat", 11.00, 10);
            //One item at $11.00 with a 10% discount.

        System.out.println(simple);
        System.out.println(discount);

        if (discount.lessThan(simple))
            System.out.println("Discounted item is cheaper.");
        else
            System.out.println("Discounted item is not cheaper.");

        mySale regularPrice = new mySale("cup holder", 9.90); //One item at $9.90.
        myDiscountSale specialPrice = new myDiscountSale("cup holder", 11.00, 10);
            //One item at $11.00 with a 10% discount.

        System.out.println(regularPrice);
        System.out.println(specialPrice);

        if (specialPrice.equalDeals(regularPrice))
            System.out.println("Deals are equal.");
        else
            System.out.println("Deals are not equal.");
    }
}

```

Οι **lessThan** και **equalDeals** κληρονομούνται από την **mySale**

Με το μηχανισμό του **late binding** στην κλήση τους ξέρουμε ότι το αντικείμενο που τις καλεί είναι ΤΥΠΟΥ **myDiscountSale**

Ξέρουμε λοιπόν ότι όταν εκτελούμε τον κώδικα της **lessThan** και **equalDeals** η μέθοδος **bill()** που θα πρέπει να καλέσουμε είναι αυτή της **myDiscountSale** ενώ για το **otherSale.bill()** είναι αυτή της **mySale**

ΑΦΗΡΗΜΕΝΕΣ ΚΛΑΣΕΙΣ ΚΑΙ INTERFACES

Ένα διαφορετικό πρόβλημα

- Ας υποθέσουμε ότι στην **Employee** θέλουμε να προσθέσουμε μια μέθοδο που ελέγχει αν δύο υπάλληλοι έχουν τον ίδιο μισθό (ανεξάρτητα αν είναι ωρομίσθιοι, ή πλήρους απασχόλησης)
- Η συνάρτηση είναι απλή:

```
public boolean sameSalary(Employee other)
{
    if (this.getPay() == other.getPay()) {
        return true;
    }
    return false
}
```

- Το **πρόβλημα**: Που θα την ορίσουμε?
 - Ιδανικά στην **Employee**, αλλά η **Employee** δεν έχει συνάρτηση **getPay()**
 - Αν την ορίσουμε στην **HourlyEmployee**, ή στην **SalariedEmployee**, δεν μπορούμε να περάσουμε όρισμα **Employee** εφόσον δεν έχει μέθοδο **getPay()**

Αφηρημένες μέθοδοι

- Η λύση είναι να ορίσουμε την `getPay()` ως **αφηρημένη μέθοδο** (`abstract method`) της `Employee`.
 - `public abstract double getPay();`
 - Μια αφηρημένη μέθοδος **δηλώνεται** σε μία κλάση αλλά **ορίζεται** στις παράγωγες κλάσεις.
 - Χρησιμοποιούμε τη δεσμευμένη λέξη **abstract** για να δηλώσουμε ότι μια μέθοδος είναι αφηρημένη.
 - Η δήλωση μιας αφηρημένης μεθόδου **δεν έχει κώδικα** οπότε η εντολή τερματίζει με το **;**
 - Οι αφηρημένες μέθοδοι πρέπει να είναι **public** (ή `protected`), όχι `private`.

Αφηρημένες κλάσεις

- Οι κλάσεις που περιέχουν μια αφηρημένη μέθοδο ορίζονται **υποχρεωτικά** ως **αφηρημένες κλάσεις** (**abstract classes**)
 - `public abstract class Employee { ... }`
- **Δεν μπορούμε** να δημιουργήσουμε **αντικείμενα** μιας **αφηρημένης κλάσης**
 - Μια αφηρημένη κλάση χρησιμοποιείται μόνο για να δημιουργούμε **παράγωγες κλάσεις**.
 - Στην περίπτωση μας δεν χρειαζόμαστε αντικείμενα τύπου Employee. Ένας υπάλληλος θα είναι είτε ωρομίσθιος, είτε μόνιμος.
- Οι **παράγωγες** κλάσεις μιας αφηρημένης κλάσης θα **πρέπει πάντα** να **ορίζουν** τις **αφηρημένες μεθόδους**
 - **Εκτός** αν είναι και αυτές **αφηρημένες**.
- Μια κλάση (ή μέθοδος) που δεν είναι αφηρημένη λέγεται **ενυπόστατη** (**concrete**)

```
public abstract class Employee
```

Ορισμός της αφηρημένης κλάσης

```
{
```

```
    private String name;
```

```
    private Date hireDate;
```

Ορισμός της αφηρημένης μεθόδου

```
    public abstract double getPay();
```

```
    public boolean samePay(Employee other){
```

```
        return (this.getPay() == other.getPay());
```

```
    }
```

Χρήση της αφηρημένης μεθόδου και της αφηρημένης κλάσης

```
    public Employee( ) { ... }
```

```
    public Employee(String theName, Date theDate) { ... }
```

```
    public Employee(Employee originalObject) { ... }
```

```
    public String getName( ) { ... }
```

```
    public void setName(String newName) { ... }
```

```
    public Date getHireDate( ) { ... }
```

```
    public void setHireDate(Date newDate) { ... }
```

```
    public String toString()
```

Όταν καλέσουμε την **samePay** θα την καλέσουμε με ένα αντικείμενο μιας από τις παράγωγες κλάσεις.

```
}
```



```

public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee( ) { ... }

    public HourlyEmployee(String theName, Date theDate,
        double theWageRate, double theHours) { ... }

    public HourlyEmployee(HourlyEmployee originalObject) { ... }

    public double getRate( ) { ... }
    public void setRate(double newWageRate) { ... }

    public double getHours( ) { ... }
    public void setHours(double hoursWorked) { ... }

    public double getPay( ) {
        return wageRate*hours;
    }

    public String toString( ){ ... }
}

```

Εφόσον η κλάση HourlyEmployee παράγεται από αφηρημένη κλάση και η ίδια δεν είναι αφηρημένη, πρέπει **υποχρεωτικά** να ορίσει την αφηρημένη μέθοδο getPay

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { ... }

    public SalariedEmployee(String theName,
                             Date theDate, double theSalary) { ... }

    public SalariedEmployee(SalariedEmployee originalObject ) { ... }

    public double getSalary( ) { ... }
    public void setSalary(double newSalary) { ... }

    public double getPay( )
    {
        return salary/12;
    }

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

Εφόσον η κλάση SalariedEmployee παράγεται από αφηρημένη κλάση και η ίδια δεν είναι αφηρημένη, πρέπει **υποχρεωτικά** να ορίσει την αφηρημένη μέθοδο getPay

```
public class Example
{
    public static void main(String args[]) {
        HourlyEmployee A = new HourlyEmployee("Alice",
            new Date(4,18,2013), 10, 100);
        SalariedEmployee B = new SalariedEmployee("Bob",
            new Date(4,17,2013), 12000);
        if (A.samePay(B)) {
            System.out.println("The two employees
                earn the same amount per month");
        }
        else{
            System.out.println("The two employees do NOT
                earn the same amount per month");
        }
    }
}
```

```

public class Example
{
    public static void main(String args[]) {
        Employee A = new HourlyEmployee("Alice",
            new Date(4,18,2013), 10, 100);
        Employee B = new SalariedEmployee("Bob",
            new Date(4,17,2013), 12000);
        if (A.samePay(B)) {
            System.out.println("The two employees
                earn the same amount per month");
        }
        else{
            System.out.println("The two employees do NOT
                earn the same amount per month");
        }
    }
}

```

Μπορούμε να ορίσουμε **μεταβλητές αφηρημένης κλάσης**. Θα πρέπει να όμως να τους αναθέσουμε **αντικείμενα** μιας από τις **παράγωγες ενυπόστατες κλάσεις**. Δεν μπορούμε να ορίσουμε ένα αντικείμενο της αφηρημένης κλάσης.

```
public class Example
{
    public static void main(String args[]){
        HourlyEmployee A = new HourlyEmployee("Alice",
            new Date(4,18,2013), 10, 100);
        SalariedEmployee B = new SalariedEmployee("Bob",
            new Date(4,17,2013), 12000);
        compareAndPrint(A,B)
    }

    private static void compareAndPrint(Employee A, Employee B){
        if (A.samePay(B)){
            System.out.println("The two employees
                earn the same amount per month");
        }
        else{
            System.out.println("The two employees do NOT
                earn the same amount per month");
        }
    }
}
```

Αφηρημένες κλάσεις

- **Αφηρημένες κλάσεις** είναι οι κλάσεις που περιέχουν **αφηρημένες μεθόδους**
 - Η **υλοποίηση** των αφηρημένων μεθόδων μετατίθεται στις μη αφηρημένες (**ενυπόστατες – concrete**) κλάσεις που είναι **απόγονοι** μιας **αφηρημένης κλάσης**.
 - Η υλοποίηση είναι **υποχρεωτική**. Άρα έτσι εξασφαλίζουμε ότι μια concrete κλάση θα έχει την μέθοδο που θέλουμε.
- Οι αφηρημένες κλάσεις εκτός από αφηρημένες μεθόδους έχουν και **πεδία** και **ενυπόστατες μεθόδους**.
 - Κληρονομούν επιπλέον **χαρακτηριστικά** στους απογόνους τους, όχι μόνο τις αφηρημένες μεθόδους.

Interfaces

- Ένα **interface** είναι μια ακραία μορφή αφηρημένης κλάσης
 - Ένα interface έχει **μόνο δηλώσεις** μεθόδων.
 - Το interface ορίζει μια **απαραίτητη λειτουργικότητα** που θέλουμε.

Παραδείγματα

```
public interface MovingObject
{
    public void move();
}
```

```
public interface ElectricObject
{
    public boolean powerOn();
    public boolean powerOff();
}
```


Interfaces

- Μία κλάση υλοποιεί το interface.
 - Η κλάση μπορεί να είναι και αφηρημένη κλάση
- Μια κλάση μπορεί να υλοποιεί πολλαπλά interfaces
 - Αλλά δεν μπορεί να κληρονομεί από πολλαπλές κλάσεις

Παραδείγματα

```
public class Car implements MovingObject
{
    ...
}
```

```
public class ElectricCar
    implements MovingObject, ElectricObject
{
    ...
}
```

```
public abstract class Vehicle implements MovingObject
{
    public abstract void move();
}
```

```
public class ElectricCar
    extends Vehicle, implements ElectricObject
{
    ...
}
```

Interfaces

- Ένα Interface μπορεί να κληρονομεί από ένα άλλο interface

```
public interface ElectricMovingObject
    extends MovingObject
{
    public boolean powerOn();

    public boolean powerOff();
}
```

Κληρονομεί και τις μεθόδους της διεπαφής MovingObject

Interfaces vs αφηρημένες κλάσεις

- Τα **interfaces** είναι χρήσιμα όταν θέλουμε να ορίσουμε αντικείμενα που ορίζονται μόνο από κάποια **υψηλού επιπέδου λειτουργικότητα** ενώ κατά τα άλλα μπορεί να είναι πολύ διαφορετικά μεταξύ τους
 - Έχουν το ίδιο interface – ένα κινούμενο αντικείμενο μπορεί να κινείται
 - Δεν ξέρουμε πως, σε πόσες διαστάσεις, με τι ταχύτητα κλπ.
- Μια **αφηρημένη κλάση** υποθέτει ότι τα αντικείμενα που θα ορίσουμε έχουν πολλά περισσότερα **κοινά χαρακτηριστικά**
 - Κοινά πεδία πάνω στα οποία μπορούμε να υλοποιήσουμε και κοινές μεθόδους.

Αφηρημένοι Τύποι Δεδομένων

- Τα interfaces μπορούμε να τα δούμε και σαν **Αφηρημένους Τύπους Δεδομένων**
- Π.χ., μία **στοίβα** απαιτεί συγκεκριμένες λειτουργίες από τις κλάσεις που την υλοποιούν
 - Push
 - Pop
 - IsEmpty
 - Top
- Ανάλογα με τον τύπο των δεδομένων που θα κρατάει η στοίβα μπορούμε να ορίσουμε διαφορετικές **υλοποιήσεις**
 - Υπάρχει και άλλος τρόπος να το κάνουμε αυτό όμως όπως θα δούμε παρακάτω

Παράδειγμα: Το interface myComparable

- Το interface **myComparable** ορίζει interface για αντικείμενα τα οποία μπορούν να **συγκριθούν** μεταξύ τους
 - Υπάρχει στην Java το interface Comparable αλλά είναι λίγο διαφορετικό
- Ορίζει την μέθοδο
 - `public int compareTo(myComparable other);`
- Σημασιολογία:
 - Αν η μέθοδος επιστρέψει **αρνητικό αριθμό** τότε το αντικείμενο **this** είναι **μικρότερο** από το αντικείμενο **other**
 - Αν η μέθοδος επιστρέψει **μηδέν** τότε το αντικείμενο **this** είναι **ίσο** με το αντικείμενο **other**
 - Αν η μέθοδος επιστρέψει **θετικό αριθμό** τότε το αντικείμενο **this** είναι **μεγαλύτερο** από το αντικείμενο **other**

Interface myComparable

```
public interface myComparable
{
    public int compareTo(myComparable other);
}
```

Εφαρμογή

- Μπορούμε να ορίσουμε μια μέθοδο `sort` η οποία να μπορεί να εφαρμοστεί σε πίνακες με οποιαδήποτε μορφής αντικείμενα

```
public static void sort(myComparable[] array) {  
    for (int i = 0; i < array.length; i ++){  
        myComparable minElement = array[i];  
        for (int j = i+1; j < array.length; j ++){  
            if (minElement.compareTo(array[j]) > 0){  
                minElement = array[j];  
                array[j] = array[i];  
                array[i] = minElement;  
            }  
        }  
    }  
}
```

Μπορεί να εφαρμοστεί σε **οποιαδήποτε** αντικείμενα που υλοποιούν το interface `myComparable`


```
import java.util.Scanner;

class Person implements myComparable
{
    private String name;
    private int number;

    public Person() {
        System.out.println("enter name and number:");
        Scanner input = new Scanner(System.in);
        name = input.next(); number = input.nextInt();
    }

    public String toString() {
        return name + " " + number;
    }

    public int compareTo(myComparable other) {
        Person otherPerson = (Person) other;
        if (number < otherPerson.number) {
            return -1;
        } else if (number == otherPerson.number) {
            return 0;
        } else { return 1;}
    }
}
```

Χρήση του DownCasting

```
public class ComparableExample
{
    public static void main(String[] args){
        Person[] array = new Person[5];
        for (int i = 0; i < array.length; i ++){
            array[i] = new Person();
        }
        sort(array);
        System.out.println();
        for (int i = 0; i < array.length; i ++){
            System.out.println(array[i]);
        }
    }

    public static void sort(myComparable[] array){
        for (int i = 0; i < array.length; i ++){
            myComparable minElement = array[i];
            for (int j = i+1; j < array.length; j ++){
                if (minElement.compareTo(array[j]) > 0){
                    minElement = array[j];
                    array[j] = array[i];
                    array[i] = minElement;
                }
            }
        }
    }
}
```

Επέκταση

- Τι γίνεται αν αντί για Persons θέλουμε να συγκρίνουμε σπίτια?
 - Ένα σπίτι (House) έχει διεύθυνση και μέγεθος
 - Θέλουμε να ταξινομήσουμε με βάση το μέγεθος

Βρείτε τα λάθη

- Στο πρόγραμμα στην επόμενη διαφάνεια υπάρχουν διάφορα λάθη
 - Ποια είναι?

```
public abstract class Vehicle
{
    private int position = 0;

    public Vehicle(int pos){
        position = pos;
    }

    public abstract void move();

    public void print()
    {
        System.out.println("position = "
            + position);
    }
}
```

```
public class Example
{
    public static void main(String[] args){
        Vehicle[] V = new Vehicle[3];
        V[0] = new Car(0,100);
        V[1] = new Bike();
        V[2] = new Vehicle(0);
        V[0].drive(); V[0].print();
        V[1].move(); V[1].print();
        int gas = V[0].getGas();
    }
}
```

```
public class Car extends Vehicle
{
    private int gas;

    public Car(int pos, int gas){
        position = pos;
        this.gas = gas;
    }

    public void drive(){
        position += 10;
        gas -= 10;
    }

    public int getGas(){
        return gas;
    }

    public void print(){
        super.print();
        System.out.println("gas =" + gas);
    }
}
```

```
public class Bike extends Vehicle
{
    public void move(){
        position ++;
    }
}
```

```
public abstract class Vehicle
{
    private int position = 0;

    public Vehicle(int pos){
        position = pos;
    }

    public abstract void move();

    public void print()
    {
        System.out.println("position = "
            + position);
    }
}
```

```
public class Example
{
    public static void main(String[] args){
        Vehicle[] V = new Vehicle[3];
        V[0] = new Car(0,100);
        V[1] = new Bike();
        V[2] = new Vehicle(0);
        V[0].drive(); V[0].print();
        V[1].move(); V[1].print();
        int gas = V[0].getGas();
    }
}
```

```
public class Car extends Vehicle
{
    private int gas;

    public Car(int pos, int gas){
        position = pos;
        this.gas = gas;
    }

    public void drive(){
        position += 10;
        gas -= 10;
    }

    public int getGas(){
        return gas;
    }

    public void print(){
        super.print();
        System.out.println("gas =" + gas);
    }
}
```

```
public class Bike extends Vehicle
{
    public void move(){
        position ++;
    }
}
```

```

public abstract class Vehicle
{
    protected int position = 0;

    public Vehicle() {
    }

    public Vehicle(int pos) {
        position = pos;
    }

    public int getPosition() {
        return position;
    }

    public void setPosition(int pos) {
        position = pos;
    }

    public abstract void move();

    public void print()
    {
        System.out.println("position = "
            + position);
    }
}

```

Το πεδίο position πρέπει να είναι **protected** εφόσον το χρησιμοποιούν και οι παράγωγες κλάσεις ή να ορίσουμε **getPosition** και **setPosition** μεθόδους

Πρέπει να ορίσουμε και ένα κενό **constructor**, ή να καλούμε την **super** μέσα στις παράγωγες κλάσεις.

```

public class Car extends Vehicle
{
    private int gas;

    public Car(int pos, int gas){
        position = setPosition(pos);
        this.gas = gas;
    }

    public void move(){
        setPosition(getPosition() + 10);
        gas -= 10;
    }

    public int getGas(){
        return gas;
    }

    public void print(){
        super.print();
        System.out.println("gas =" + gas);
    }
}

```

Ο **constructor** δουλεύει μόνο αν έχουμε constructor χωρίς ορίσματα στην **Vehicle**. Αλλιώς χρειαζόμαστε αυτό τον constructor:

```

public Car(int pos, int gas){
    super(pos);
    this.gas = gas;
}

```

Η Car πρέπει να υλοποιεί την μέθοδο **move**


```
public class Bike extends Vehicle
{
    public void move() {
        position ++;
    }
}
```

Ο **constructor** (ή μάλλον η έλλειψη του) δουλεύει μόνο αν έχουμε constructor χωρίς ορίσματα στην **Vehicle**. Αλλιώς χρειαζόμαστε αυτό τον constructor:

```
public Bike() {
    super(0);
}
```

```

public class Example
{
    public static void main(String[] args) {
        Vehicle[] V = new Vehicle[2];
        V[0] = new Car(0,100);
        V[1] = new Bike();
        //V[2] = new Vehicle(0);
        V[0].move(); V[0].print();
        V[1].move(); V[1].print();
        int gas = ((Car)V[0]).getGas();
    }
}

```

Δεν μπορούμε να δημιουργήσουμε αντικείμενο τύπου **Vehicle** γιατί είναι αφηρημένη κλάση.

Η **Vehicle** δεν έχει μέθοδο `getGas`.
Για να την καλέσουμε θα πρέπει να κάνουμε **downcast** το αντικείμενο `V[0]` σε `Car`.

Ερωτήσεις:

- Υπάρχει πρόβλημα με την εντολή `Vehicle[] V = new Vehicle[2];` ?
- Ποια `print` καλείται για το αντικείμενο `V[0]`? Ποια για το `V[1]`? Γιατί?
- Τι θα τυπώσει το πρόγραμμα?

Υπάρχει κάποιο λάθος σε αυτό τον ορισμό?

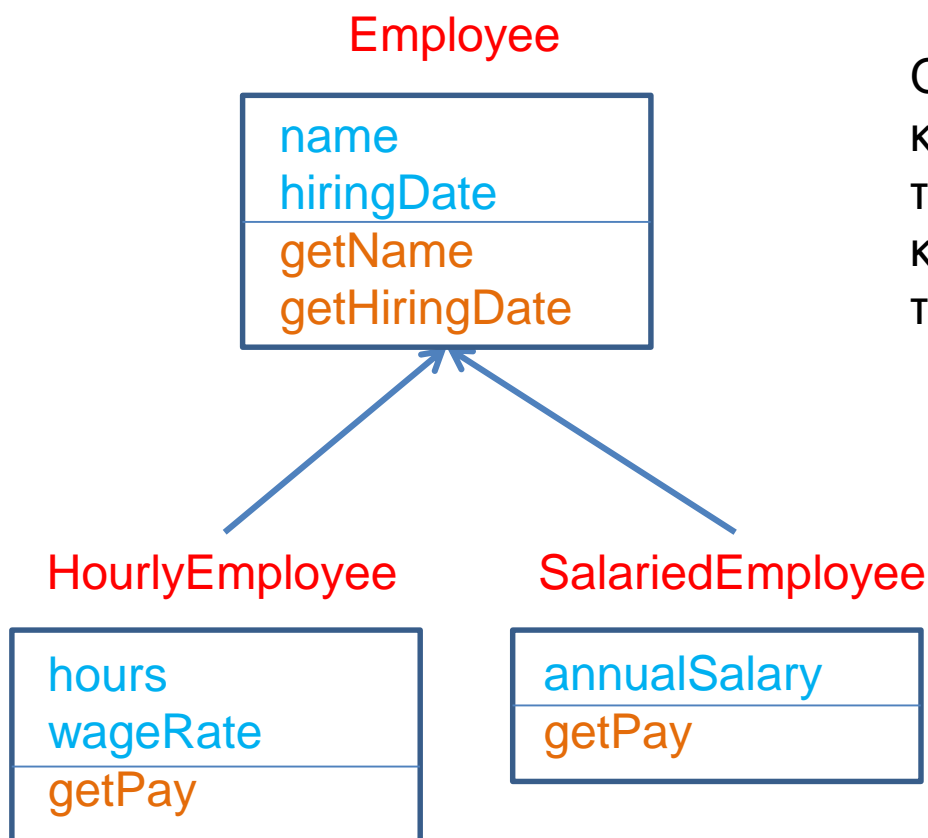
```
public abstract class EngineVehicle extends Vehicle
{
    protected int gas;

    public EngineVehicle(int pos, int gas) {
        super(pos);
        this.gas = gas;
    }
}
```

Όχι. Εφόσον η EngineVehicle είναι αφηρημένη δεν χρειάζεται να ορίσουμε την αφηρημένη μέθοδο move

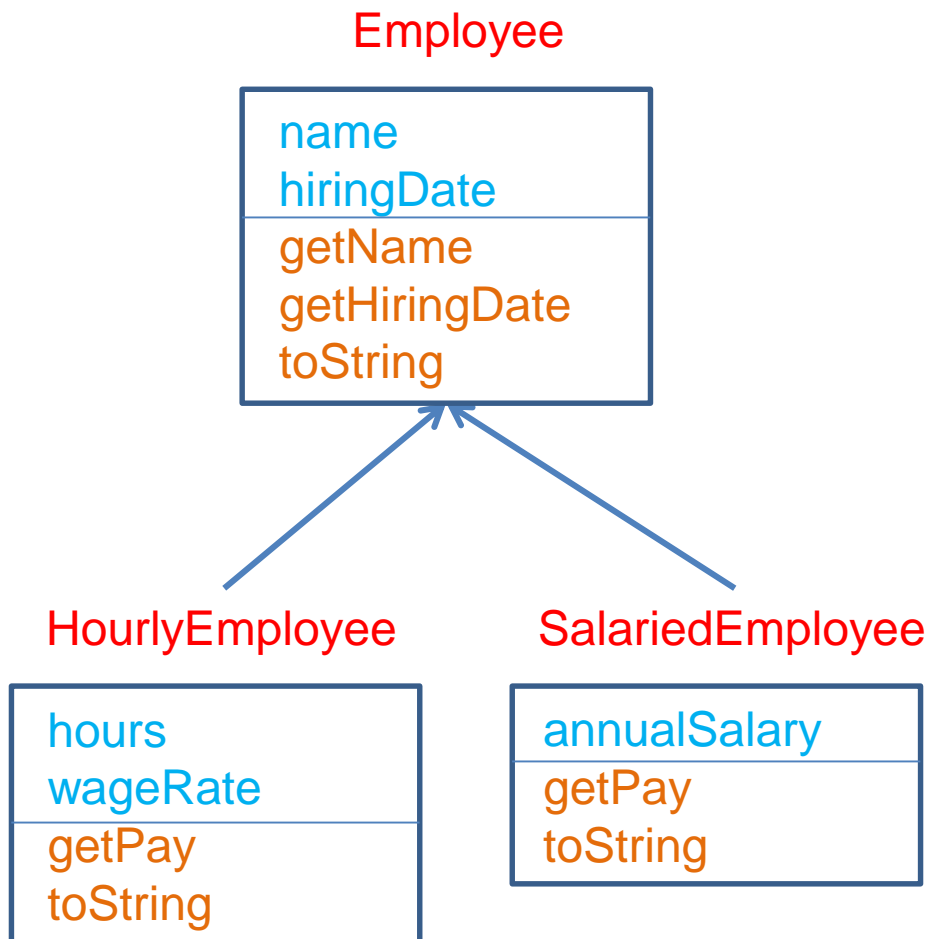
10. ΠΑΡΑΔΕΙΓΜΑΤΑ ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑΣ

Κληρονομικότητα



Οι παράγωγες κλάσεις κληρονομούν τα πεδία και τις μεθόδους της βασικής κλάσης και έχουν και δικά τους πεδία και μεθόδους

Late Binding

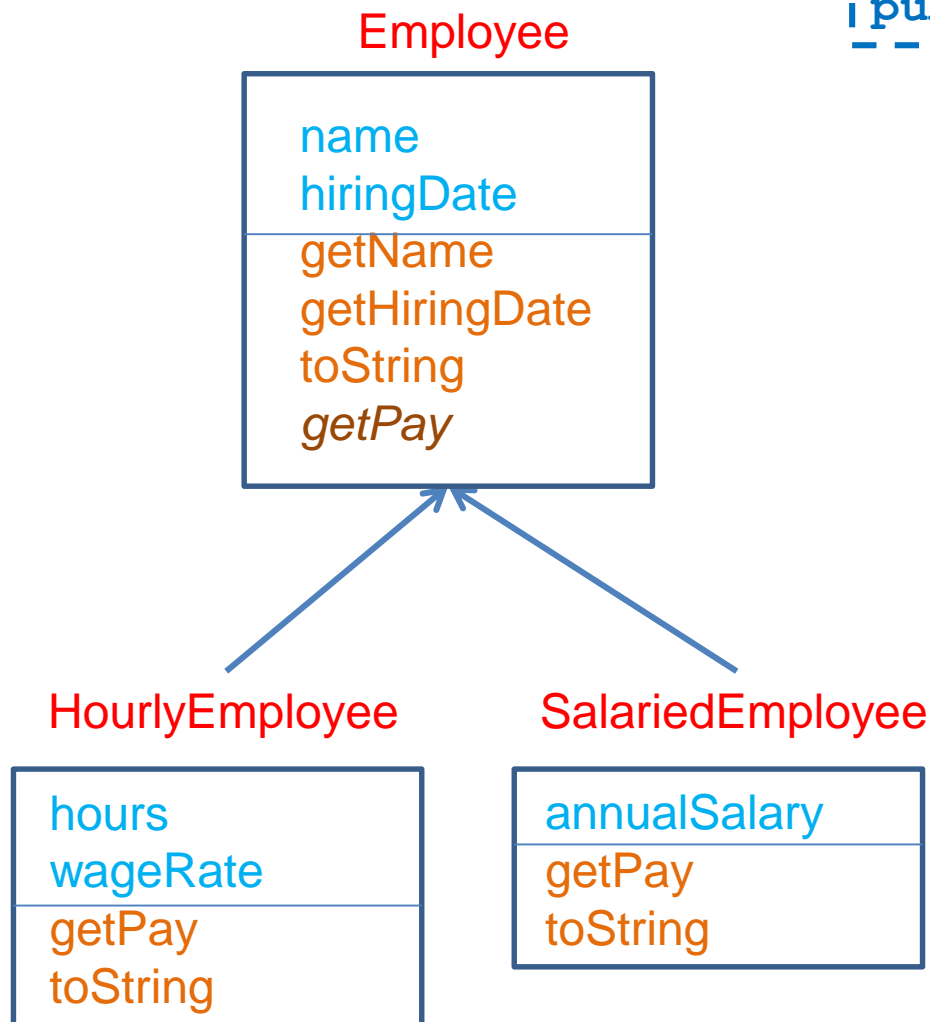


```
Employee e;  
e = new HourlyEmployee();  
System.out.println(e);  
e = new SalariedEmployee();  
System.out.println(e);
```

Late Binding:

Ο κώδικας που εκτελείται για την `toString()` εξαρτάται από την κλάση του αντικειμένου την ώρα της κλήσης (`HourlyEmployee` ή `SalariedEmployee`) και όχι την ώρα της δήλωσης (`Employee`)

Αφηρημένες κλάσεις



```
public abstract double getPay();
```

Μια **αφηρημένη μέθοδος** δηλώνεται σε μια γενική κλάση και **ορίζεται** σε μια πιο εξειδικευμένη κλάση

Οι κλάσεις με αφηρημένες μεθόδους είναι **αφηρημένες κλάσεις**.

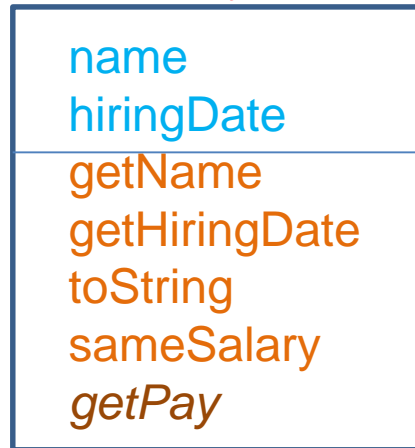
Δεν μπορούμε να **δημιουργήσουμε** αντικείμενα αφηρημένων κλάσεων.

- Δηλαδή **δεν μπορούμε** να κάνουμε `new Employee()` εφόσον η Employee είναι αφηρημένη

Οι παράγωγες **ενυπόστατες** κλάσεις πρέπει να **υλοποιούν** τις αφηρημένες μεθόδους.

Αφηρημένες κλάσεις

Employee



```
public boolean sameSalary(Employee other)
{
    if(this.getPay() == other.getPay()){
        return true;
    }
    return false
}
```

HourlyEmployee

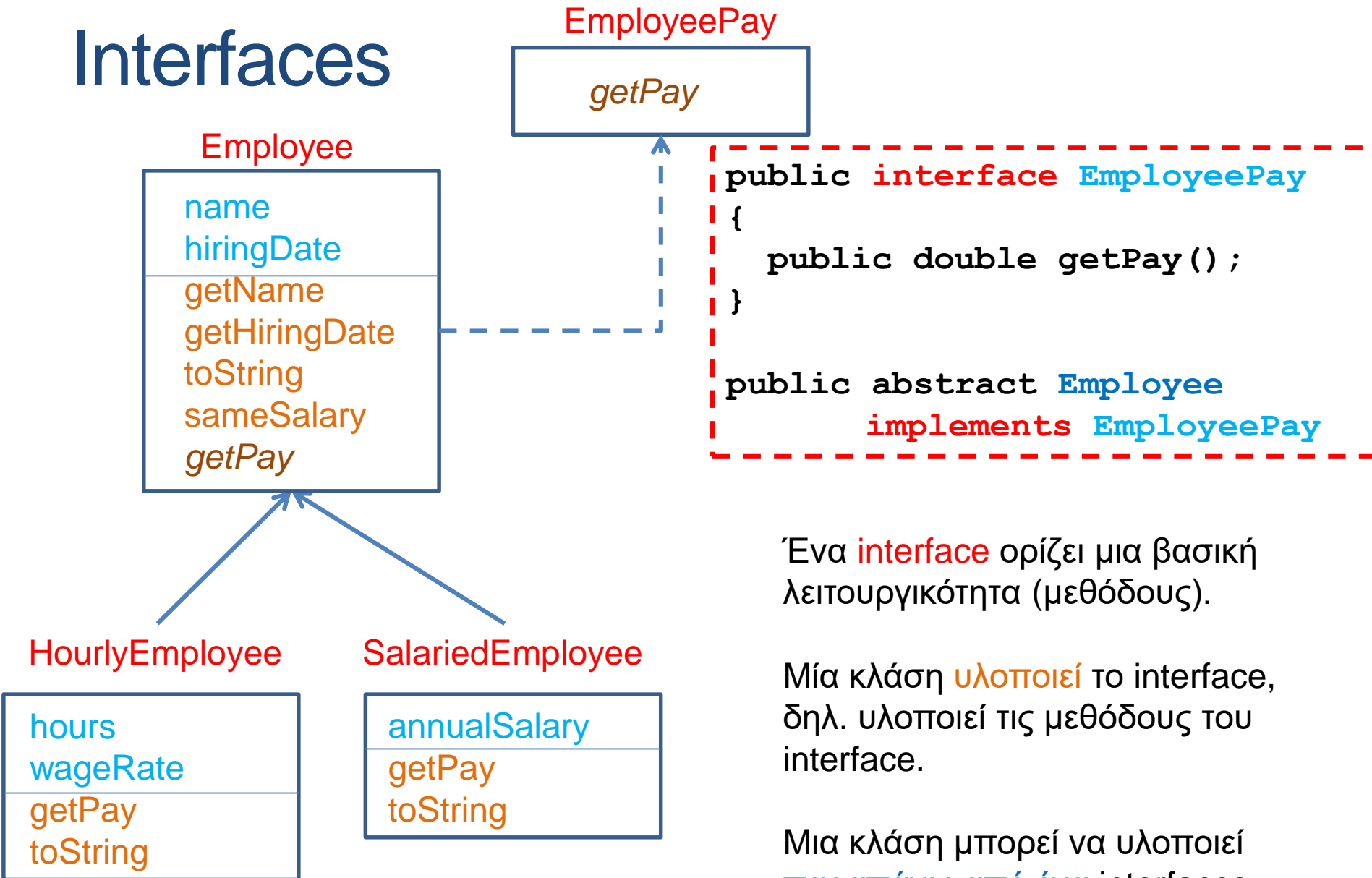
hours wageRate
getPay toString

SalariedEmployee

annualSalary
getPay toString

Μια **αφηρημένη μέθοδος** μπορεί να χρησιμοποιηθεί μέσα στις ενυπόστατες μεθόδους της αφηρημένης κλάσης

Interfaces



Ένα **interface** ορίζει μια βασική λειτουργικότητα (μεθόδους).

Μία κλάση **υλοποιεί** το interface, δηλ. υλοποιεί τις μεθόδους του interface.

Μια κλάση μπορεί να υλοποιεί **παραπάνω από ένα** interfaces

ΠΑΡΑΔΕΙΓΜΑ ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑΣ I

Ένα μεγάλο παράδειγμα

- Θέλουμε να φτιάξουμε ένα πρόγραμμα που διαχειρίζεται το **πορτοφόλιο (portfolio)** ενός χρηματιστή. Το portfolio έχει **μετοχές (stocks)**, μετοχές που δίνουν **μέρισμα (divident stocks)**, **αμοιβαία κεφάλαια (mutual funds)**, και **χρήματα (cash)**. Για κάθε μια από αυτές τις **αξίες (assets)** θέλουμε να **υπολογίζουμε** την τωρινή της **αποτίμηση (market value)** και το **κέρδος (profit)** που μας δίνει. Μετά θέλουμε να υπολογίσουμε τη συνολική αξία του πορτοφολίου και το συνολικό κέρδος

Λεπτομέρειες

- **Cash:** Δεν μεταβάλλεται η αξία του, δεν έχει κέρδος
- **Stocks:** Η αξία του είναι ίση με τον αριθμό των μετοχών επί την αξία της μετοχής. Το κέρδος είναι η διαφορά της τωρινής αποτίμησης με το **κόστος αγοράς**
- **Mutual Funds:** Παρόμοια με τα Stocks αλλά ο αριθμός των μετοχών που μπορούμε να έχουμε είναι **πραγματικός αριθμός** αντί για ακέραιος
- **Dividend Stocks:** Όμοια με τα Stocks αλλά στο κέρδος προσθέτουμε και τα **μερίσματα**

Stock

symbol number: int cost current price
getMarketValue getProfit

MutualFunds

symbol number: double cost current price
getMarketValue getProfit

DividendStock

symbol number: int cost current price dividends
getMarketValue getProfit

Cash

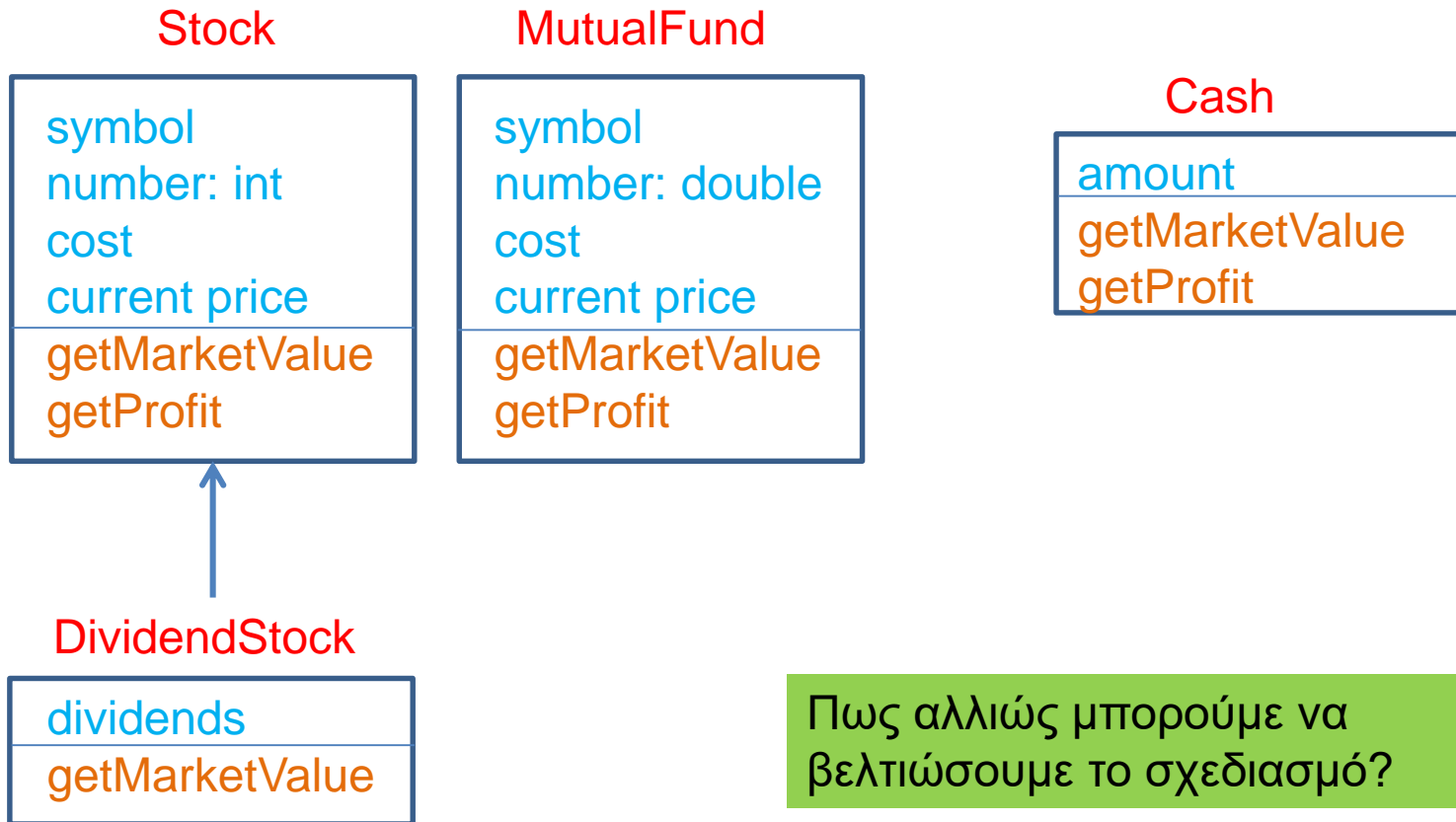
amount
getMarketValue getProfit

Πως μπορούμε να βελτιώσουμε το σχεδιασμό των κλάσεων?

Σχεδιασμός

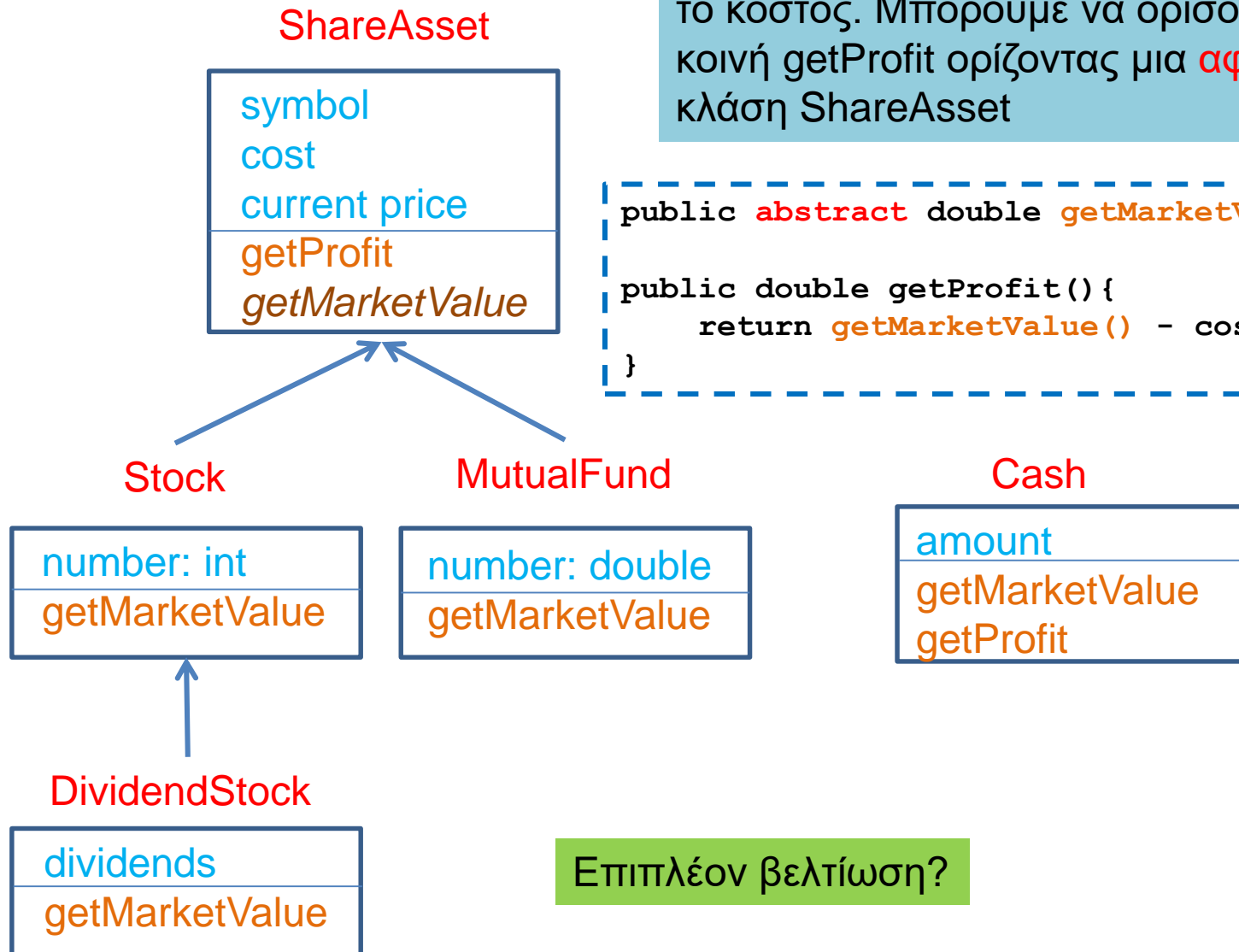
- Βλέπουμε ότι υπάρχουν διάφορα **κοινά στοιχεία** μεταξύ των διαφόρων οντοτήτων που μας ενδιαφέρουν
 - Χρειαζόμαστε για κάθε **asset** μια συνάρτηση που να μας δίνει το **market value** και μία που να υπολογίζει το **profit**
 - Για τα share assets (stocks, dividend stocks, mutual funds) το κέρδος είναι η **διαφορά** της **τωρινής τιμής** με το **κόστος**
 - Η τιμή των dividend stocks υπολογίζεται όπως αυτή την απλών stocks απλά προσθέτουμε και το μέρισμα

Η DividentStock έχει τα ίδια χαρακτηριστικά με την Stock και απλά αλλάζει ο τρόπος που υπολογίζεται η αποτίμηση ώστε να προσθέτει τα dividends



Πως αλλιώς μπορούμε να βελτιώσουμε το σχεδιασμό?

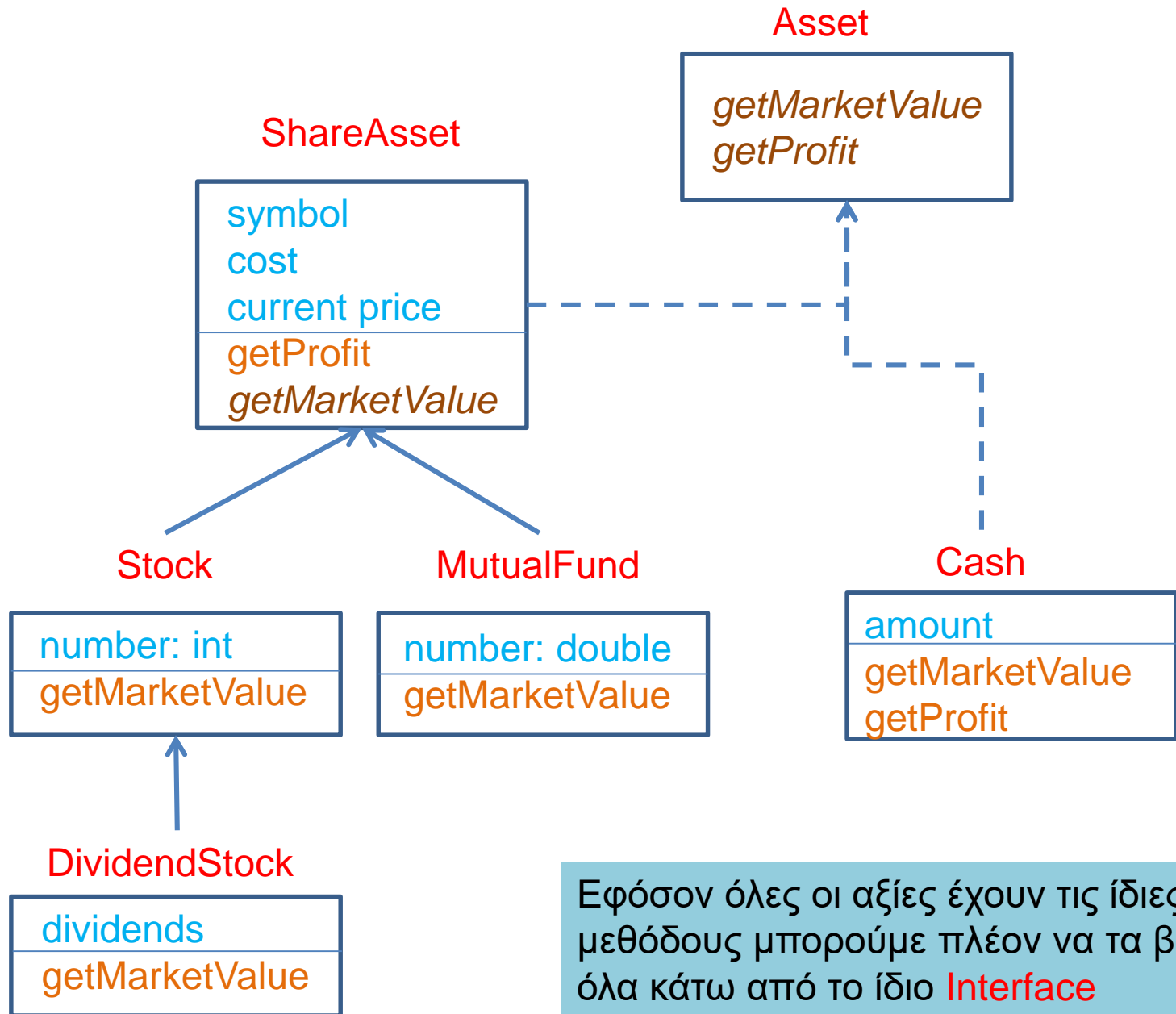
Η `getProfit` είναι ουσιαστικά η ίδια για όλα τα shares: τωρινή αποτίμηση μείον το κόστος. Μπορούμε να ορίσουμε μια κοινή `getProfit` ορίζοντας μια αφηρημένη κλάση `ShareAsset`



```
public abstract double getMarketValue();

public double getProfit(){
    return getMarketValue() - cost;
}
```

Επιπλέον βελτίωση?



Εφόσον όλες οι αξίες έχουν τις ίδιες μεθόδους μπορούμε πλέον να τα βάλουμε όλα κάτω από το ίδιο **Interface**

```
public interface Asset
{
    public double getMarketValue();

    public double getProfit();
}
```

```
public abstract class ShareAsset implements Asset
{
    private String symbol;
    private double cost = 0;
    private double currentPrice;

    public ShareAsset(String symbol, double price){
        this.symbol = symbol;
        currentPrice = price;
    }

    public abstract double getMarketValue();

    public double getProfit(){
        return getMarketValue() - cost;
    }

    public void setCost(double cost){ this.cost += cost;}

    public void setCurrentPrice(double price){currentPrice = price;}

    public double getCost(){ return cost; }

    public double getCurrentPrice(){return currentPrice;}

    public String getSymbol(){return symbol;}
}
```

```
public class Stock extends ShareAsset
{
    private int number = 0;

    public Stock(String symbol, int number, double costPrice){
        super(symbol, costPrice);
        this.number = number;
        setCost(number*costPrice);
    }

    public Stock(String symbol, double costPrice){
        super(symbol, costPrice);
    }

    public void purchase(int number, double price){
        this.number += number;
        setCost(number*price+getCost());
    }

    public double getMarketValue () {
        return number*getCurrentPrice ();
    }

    public int getNumber(){return number;}

    public String toString(){
        return getSymbol() +": " + number + " cost:" + getCost()
            + "\nCurrent price: " + getCurrentPrice()
            + "\nMarket Value: " + getMarketValue()
            + "\nProfit: " + getProfit();
    }
}
```

```
public class MutualFund extends ShareAsset
{
    private double number = 0;

    public MutualFund(String symbol, double number, double costPrice){
        super(symbol, costPrice);
        this.number = number;
        setCost(number*costPrice);
    }

    public MutualFund(String symbol, double costPrice){
        super(symbol, costPrice);
    }

    public void purchase(double number, double price){
        this.number += number;
        setCost(number*price+getCost());
    }

    public double getMarketValue(){
        return number*getCurrentPrice();
    }

    public double getNumber(){return number;}

    public String toString(){
        return getSymbol() + ": " + number + " cost:" + getCost()
            + "\nCurrent price: " + getCurrentPrice()
            + "\nMarket Value: " + getMarketValue()
            + "\nProfit: " + getProfit();
    }
}
```

```
public class DividendStock extends Stock
{
    private double dividends = 0;

    public DividendStock(String symbol, int number, double costPrice){
        super(symbol,number, costPrice);
    }

    public DividendStock(String symbol, double costPrice){
        super(symbol,costPrice);
    }

    public void payDividends(double amountPerShare){
        dividends += amountPerShare*getNumber();
    }

    public double getMarketValue(){
        return super.getMarketValue() + dividends;
    }

    public String toString(){
        return super.toString() + "\nDividends: " + dividends;
    }
}
```

Κλήση των μεθόδων της γονικής κλάσης με χρήση της λέξης super

```
public class Cash implements Asset
{
    private double amount = 0;

    public Cash(double amount)
    {
        this.amount = amount;
    }

    public double getMarketValue() {
        return amount;
    }

    public double getProfit() {
        return 0;
    }

    public String toString() {
        return "Cash: " + amount;
    }
}
```

```
import java.util.*;
```

```
public class Portofolio  
{
```

Χρήση του Interface Asset

```
    public static void main(String[] args){
```

```
        ArrayList<Asset> myPortofolio = new ArrayList<Asset>();
```

```
        myPortofolio.add(new Cash(1000));
```

```
        DividendStock msft = new DividendStock("MSFT", 100, 39.5);
```

```
        myPortofolio.add(msft);
```

```
        msft.setCurrentPrice(40);
```

```
        msft.payDividends(0.5);
```

```
        MutualFund fund = new MutualFund("FUND", 10.5, 30);
```

```
        myPortofolio.add(fund);
```

```
        fund.setCurrentPrice(40);
```

```
        fund.purchase(3.5, 40);
```

```
        Stock appl = new Stock("APPL", 10, 100);
```

```
        myPortofolio.add(appl);
```

```
        appl.setCurrentPrice(97);
```

```
        double totalValue = 0;
```

```
        double totalProfit = 0;
```

```
        for (Asset a:myPortofolio){
```

```
            System.out.println(a+"\n");
```

```
            totalValue += a.getMarketValue();
```

```
            totalProfit += a.getProfit();
```

```
        }
```

```
        System.out.println("\nTotal value = "+ totalValue);
```

```
        System.out.println("Total profit = "+ totalProfit);
```

```
    }
```

```
}
```

Χρήση των μεθόδων του Interface


```

public class Broker
{
    public static void main(String[] args){
        Asset[] portofolio = new Asset[4];
        portofolio[0] = new Cash(20000);
        portofolio[1] = new Stock("GOOG", 100, 800);
        portofolio[2] = new MutualFund("Fund", 10.5, 54.3);
        portofolio[3] = new DividendStock("APPL",200, 900);

        Stock goog = (Stock)portofolio[1];
        goog.setCurrentPrice(900);
        MutualFund fund = (MutualFund)portofolio[2];
        fund.setCurrentPrice(50);
        DividendStock appl = (DividendStock)portofolio[3];
        appl.setCurrentPrice(1000);
        appl.payDividends(0.5);

        double totalValue = 0;
        double totalProfit = 0;
        for (int i = 0; i < 4; i ++){
            System.out.println(portofolio[i]);
            totalValue += portofolio[i].getMarketValue();
            totalProfit += portofolio[i].getProfit();
        }
        System.out.println("Total Value = "+totalValue);
        System.out.println("Total profit = "+totalProfit);
    }
}

```

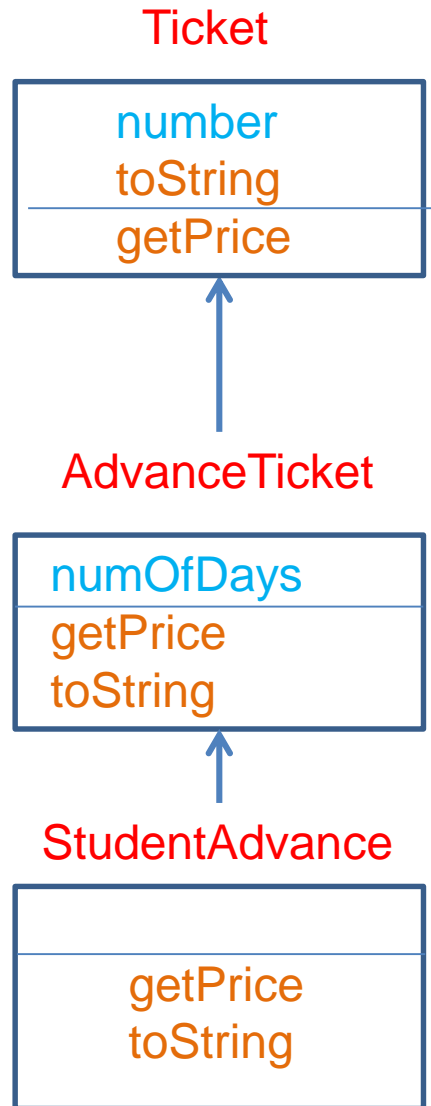
Χρειαζόμαστε downcasting για να κάνουμε χρήση των μεθόδων

ΠΑΡΑΔΕΙΓΜΑ ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑΣ II

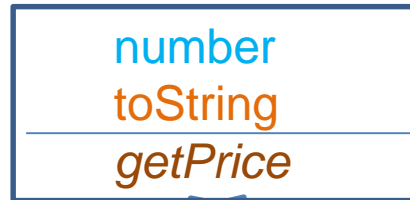
Παράδειγμα κληρονομικότητας

- Έχουμε ένα σύστημα διαχείρισης **εισιτηρίων** μιας συναυλίας. Το κάθε εισιτήριο έχει ένα **νούμερο** και **τιμή**. Η τιμή του εισιτηρίου εξαρτάται αν θα αγοραστεί στην **είσοδο** (50 ευρώ), ή θα αγοραστεί μέχρι και **10 μέρες πριν την συναυλία** (40 ευρώ), ή **πάνω από 10 μέρες πριν την συναυλία** (30 ευρώ). Τα εισιτήρια εκ των προτέρων έχουν **φοιτητική έκπτωση 50%**.
- Θέλουμε να **τυπώσουμε τα εισιτήρια** και να **υπολογίσουμε τα συνολικά έσοδα** της συναυλίας.

Ένας σχεδιασμός



Ticket

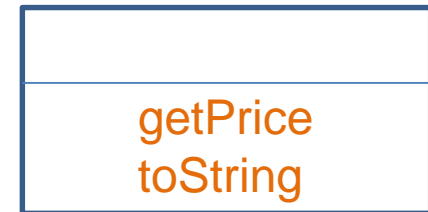


Ένας άλλος σχεδιασμός

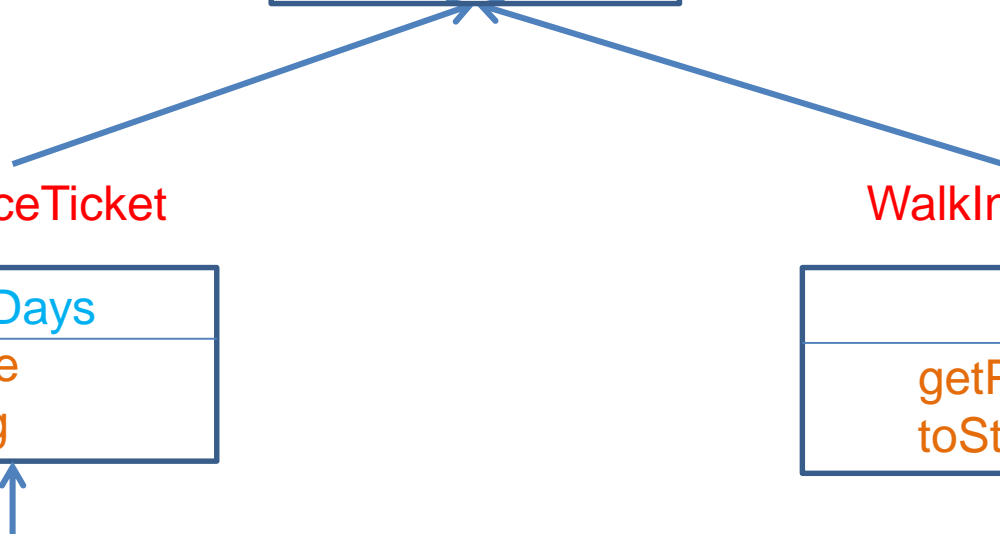
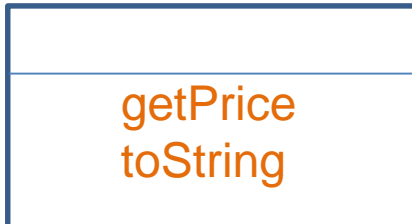
AdvanceTicket



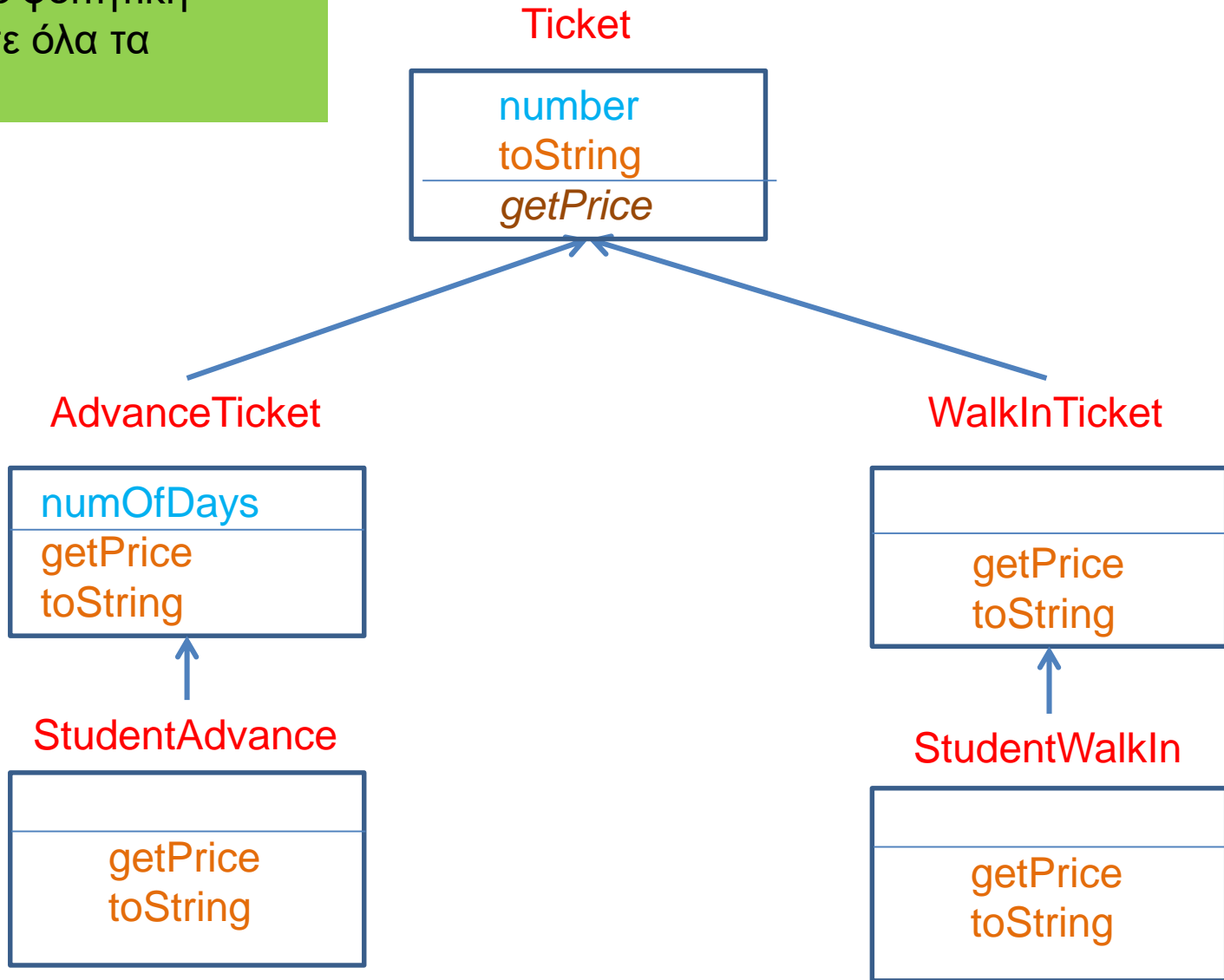
WalkInTicket



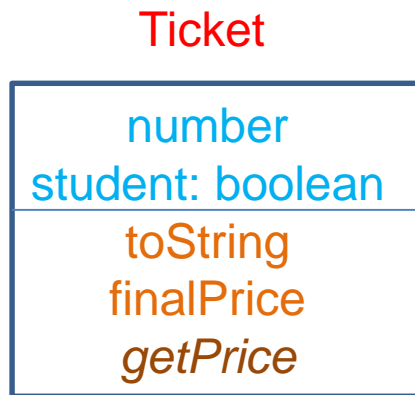
StudentAdvance



Αν θέλουμε φοιτητική έκπτωση σε όλα τα εισιτήρια?



Αν θέλουμε φοιτητική έκπτωση σε όλα τα εισιτήρια?

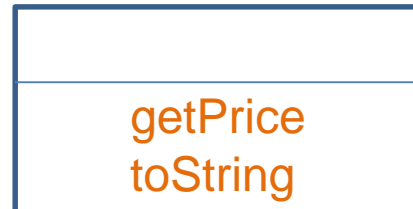


```
public abstract double getPrice();  
  
public double finalPrice()  
{  
    if (student){  
        return getPrice()*0.5;  
    }  
    return getPrice();  
}
```

AdvanceTicket



WalkInTicket



11. ΓΕΝΙΚΕΥΜΕΝΕΣ ΚΛΑΣΕΙΣ

Stack

- Θυμηθείτε πως ορίσαμε μια **στοίβα ακεραίων**

```
public class IntStackElement
{
    private int value;
    private IntStackElement next = null;

    public IntStackElement(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public IntStackElement getNext() {
        return next;
    }

    public void setNext(IntStackElement element) {
        next = element;
    }
}
```

```
public class IntStack
{
    private IntStackElement head;
    private int size = 0;

    public int pop(){
        if (size == 0){ // head == null
            System.out.println("Pop from empty stack");
            System.exit(-1);
        }
        int value = head.getValue();
        head = head.getNext();
        size--;
        return value;
    }

    public void push(int value){
        IntStackElement element = new IntStackElement(value);
        element.setNext(head);
        head = element;
        size++;
    }
}
```

Stack

- Αν θέλουμε η **στοίβα** μας να αποθηκεύει αντικείμενα της κλάσης **Person** θα πρέπει να ορίσουμε μια διαφορετική **Stack** και διαφορετική **StackElement**.

```
class PersonStackElement
{
    private Person value;
    private PersonStackElement next;

    public PersonStackElement(Person val) {
        value = val;
    }

    public void setNext(PersonStackElement element) {
        next = element;
    }

    public PersonStackElement getNext() {
        return next;
    }

    public Person getValue() {
        return value;
    }
}
```

```
public class PersonStack
{
    private PersonStackElement head;
    private int size = 0;

    public Person pop() {
        if (size == 0) { // head == null
            System.out.println("Pop from empty stack");
            System.exit(-1);
        }
        Person value = head.getValue();
        head = head.getNext();
        size--;
        return value;
    }

    public void push(Person value) {
        PersonStackElement element = new PersonStackElement(value);
        element.setNext(head);
        head = element;
        size++;
    }
}
```

Stack

- Θα ήταν πιο βολικό αν μπορούσαμε να ορίσουμε **μία μόνο** κλάση **Stack** που να μπορεί να αποθηκεύει αντικείμενα οποιουδήποτε τύπου.
 - **Πώς** μπορούμε να το κάνουμε αυτό?
- Μια λύση: Η **ObjectStack** που κρατάει αντικείμενα **Object**, την πιο γενική κλάση
- Τι πρόβλημα μπορεί να έχει αυτό?

```
class ObjectStackElement
{
    private Object value;
    private ObjectStackElement next;

    public ObjectStackElement(Object val) {
        value = val;
    }

    public void setNext(ObjectStackElement element) {
        next = element;
    }

    public ObjectStackElement getNext() {
        return next;
    }

    public Object getValue() {
        return value;
    }
}
```



```
public class ObjectStack
{
    private ObjectStackElement head;
    private int size = 0;

    public Object pop() {
        if (size == 0) { // head == null
            System.out.println("Pop from empty stack");
            System.exit(-1);
        }
        Object value = head.getValue();
        head = head.getNext();
        size--;
        return value;
    }

    public void push(Object value) {
        ObjectStackElement element = new ObjectStackElement(value);
        element.setNext(head);
        head = element;
        size++;
    }
}
```

```
public class ObjectStackTest
{
    public static void main(String[] args){
        ObjectStack stack = new ObjectStack();

        Person p = new Person("Alice", 1);
        Integer i = new Integer(10);
        String s = "a random string";

        stack.push(p);
        stack.push(i);
        stack.push(s);
    }
}
```

Δεν μπορούμε να **ελέγξουμε** τι αντικείμενα μπαίνουν στην στοίβα. Κατά την εξαγωγή θα πρέπει να γίνει **μετατροπή (downcasting)** και θέλει προσοχή να μετατρέψουμε το σωστό αντικείμενο στον σωστό τύπο.

Θέλουμε να δημιουργούμε στοίβες **συγκεκριμένου τύπου**.

Γενικευμένες (Generic) κλάσεις

- Οι γενικευμένες κλάσεις περιέχουν ένα τύπο δεδομένων **T** που ορίζεται **παραμετρικά**
- Όταν χρησιμοποιούμε την κλάση αντικαθιστούμε την παράμετρο **T** με τον **τύπο δεδομένων** (την κλάση) που θέλουμε
- ΣΥΝΤΑΚΤΙΚΟ:
 - `public class Example<T> {...}`
- Ορίζει την γενικευμένη κλάση `Example` με παράμετρο τον τύπο **T**
 - Μέσα στην κλάση ο τύπος **T** χρησιμοποιείται σαν **τύπος δεδομένων**
- Όταν χρησιμοποιούμε την κλάση `Example` αντικαθιστούμε το **T** με κάποια συγκεκριμένη **κλάση**
 - `Example<String> ex = new Example<String>();`

Ένα πολύ απλό παράδειγμα

```
public class Example<T>{
    private T data;

    public Example(T data){
        this.data = data;
    }

    public void setData(T data){
        this.data = data;
    }

    public T getData(){
        return data;
    }

    public static void main(String[] args){
        Example<String> ex = new Example<String>("hello world");
        System.out.println(ex.getData());
    }
}
```

Όταν ορίζουμε το αντικείμενο `ex` η κλάση `String` αντικαθιστά τις εμφανίσεις του `T` στον κώδικα

Ο ορισμός του constructor γίνεται χωρίς το `<T>` παρότι στην δημιουργία του αντικειμένου χρησιμοποιούμε το `<String>`

Γενικευμένη Στοίβα

- Μπορούμε τώρα να φτιάξουμε μια στοίβα για οποιοδήποτε τύπο δεδομένων

```
class StackElement<T>
{
    private T value;
    private StackElement<T> next;

    public StackElement(T val) {
        value = val;
    }

    public void setNext(StackElement<T> element) {
        next = element;
    }

    public StackElement<T> getNext() {
        return next;
    }

    public T getValue() {
        return value;
    }
}
```

```
public class Stack<T>
```

```
{
```

```
    private StackElement<T> head;
```

```
    private int size = 0;
```

```
    public T pop(){
```

```
        if (size == 0){ // head == null
```

```
            System.out.println("Pop from empty stack");
```

```
            System.exit(-1);
```

```
        }
```

```
        T value = head.getValue();
```

```
        head = head.getNext();
```

```
        size --;
```

```
        return value;
```

```
    }
```

```
    public void push(T value){
```

```
        StackElement<T> element = new StackElement<T>(value);
```

```
        element.setNext(head);
```

```
        head = element;
```

```
        size ++;
```

```
    }
```

```
}
```

```
public class StackTest
{
    public static void main(String[] args) {
        Stack<Person> personStack = new Stack<Person>();

        personStack.push(new Person("Alice", 1));
        personStack.push(new Person("Bob", 2));
        System.out.println(personStack.pop());
        System.out.println(personStack.pop());

        Stack<Integer> intStack = new Stack<Integer>();
        intStack.push(new Integer(10));
        intStack.push(new Integer(20));
        System.out.println(intStack.pop());
        System.out.println(intStack.pop());

        Stack<String> stringStack = new Stack<String>();
        stringStack.push("string 1");
        stringStack.push("string 2");
        System.out.println(stringStack.pop());
        System.out.println(stringStack.pop());
    }
}
```

Δημιουργούμε στοίβες **συγκεκριμένου τύπου**.

Πολλαπλές παράμετροι

- Μπορούμε να έχουμε πάνω από ένα παραμετρικούς τύπους

```
public class KeyValuePair<K, V>{  
    private K key;  
    private V value;  
    ...  
}
```

Παγίδες

1. Ο τύπος **T** **δεν** μπορεί να αντικατασταθεί από ένα **πρωταρχικό τύπο δεδομένων** (π.χ. int, double, boolean – πρέπει να χρησιμοποιήσουμε τα **wrapper classes** γι αυτά, Integer, Boolean, Double)
2. **Δεν** μπορούμε να ορίσουμε ένα **πίνακα** από αντικείμενα γενικευμένης κλάσης.

Π.χ., `StackElement<String>[] A =` **Δεν είναι αποδεκτό!**
`new StackElement<String>[2];`

3. **Δεν** μπορούμε να χρησιμοποιούμε τον τύπο **T** όπως οποιαδήποτε άλλη κλάση.

Π.χ., `T obj = new T();`
`T[] a = new T[10];`

Δεν είναι αποδεκτό!

Γενικευμένες κλάσεις με περιορισμούς

- Ας υποθέσουμε ότι θέλουμε να ορίσουμε μία γενικευμένη κλάση `Pair` η οποία κρατάει ένα ζεύγος από δυο αντικείμενα οποιουδήποτε τύπου.

```
public class Pair<T>{  
    private T first;  
    private T second;  
    ...  
}
```

Γενικευμένες κλάσεις με περιορισμούς

- Θέλουμε επίσης να μπορούμε να **διατάσουμε** τα ζεύγη
 - Για να γίνει αυτό θα πρέπει να υπάρχει τρόπος να **συγκρίνουμε** τα στοιχεία first και second.
 - **Περιορίζουμε** την T να **υλοποιεί** το **interface myComparable**

```
public class Pair<T extends myComparable>{  
    private T first;  
    private T second;  
  
    public void order(){  
        if (first.compareTo(second) > 0){  
            T temp = first; first = second; second = temp;  
        }  
    }  
}
```

extends όχι implements

Γενικευμένες κλάσεις με περιορισμούς

- Θέλουμε επίσης να μπορούμε να **διατάσουμε** τα ζεύγη
 - Για να γίνει αυτό θα πρέπει να υπάρχει τρόπος να **συγκρίνουμε** τα στοιχεία first και second.
 - **Περιορίζουμε** την T να **υλοποιεί** το **interface Comparable**

```
public class Pair<T extends Comparable<T>>{  
    private T first;  
    private T second;  
  
    public void order(){  
        if (first.compareTo(second) > 0){  
            T temp = first; first = second; second = temp;  
        }  
    }  
}
```

Η **Comparable<T>** της Java
Το **T** είναι ο τύπος με τον οποίο
μπορούμε να συγκρίνουμε

Γενικευμένες κλάσεις με περιορισμούς

- Μπορούμε να περιορίσουμε τον παραμετρικό τύπο να **κληρονομεί** οποιαδήποτε **κλάση**, ή οποιοδήποτε **interface** ή συνδυασμό από τα παραπάνω.

- `public class SomeClass`

- `<T extends Employee> { ... }`

Δέχεται μόνο απογόνους της Employee

- `public class SomeClass`

- `<T extends Employee & Comparable<T>>`

- `{ ... }`

Δέχεται μόνο απογόνους της Employee που υλοποιούν το interface Comparable

Μπορούμε να έχουμε πολλά διαφορετικά interfaces στους περιορισμούς, αλλά μόνο μία κλάση και αυτή θα πρέπει να προηγείται στον ορισμό

Γενικευμένες κλάσεις και κληρονομικότητα

- Μια γενικευμένη κλάση μπορεί να έχει απογόνους άλλες γενικευμένες κλάσεις.
 - Οι απόγονοι κληρονομούν και τον τύπο T.
 - `public class OrderedPair<T> extends Pair<T> { ... }`
- Δεν ορίζεται κληρονομικότητα ως προς τον παραμετρικό τύπο T
 - Δεν υπάρχει καμία σχέση μεταξύ των κλάσεων `Pair<Employee>` και `Pair<HourlyEmployee>`

Wildcard

- Αν θέλουμε να ορίσουμε ένα γενικό παραμετρικό τύπο χρησιμοποιούμε την παράμετρο μπαλαντέρ ?, η οποία αναπαριστά ένα οποιοδήποτε τύπο T.
 - Προσέξτε ότι αυτό είναι κατά τη χρήση της γενικευμένης κλάσης
- `public void someMethod(Pair<?>) { ... }`
 - Με αυτή τη δήλωση ορίζουμε μία μέθοδο που παίρνει σαν όρισμα ένα αντικείμενο Pair με τύπο T οτιδήποτε.
- Μπορούμε να περιοριστούμε σε ένα τύπο που είναι απόγονος της Employee.
- `public void someMethod(Pair<? extends Employee>) { ... }`

12. ΣΥΛΛΟΓΕΣ

ArrayList

HashSet

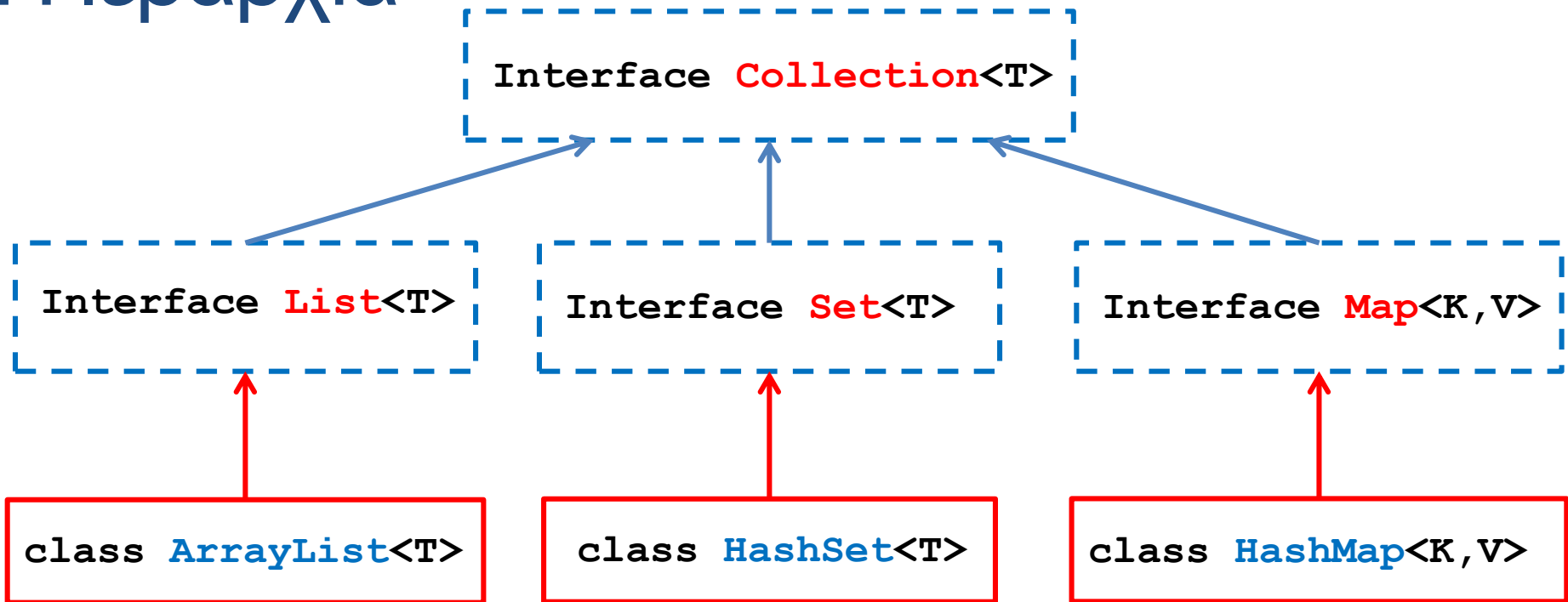
HashMap

ΣΥΛΛΟΓΕΣ

ArrayList

- Η κλάση `ArrayList<T>` είναι μια περίπτωση γενικευμένης κλάσης
 - Ένας **δυναμικός πίνακας** που ορίζεται με παράμετρο τον τύπο των αντικειμένων που θα κρατάει.
- Η `ArrayList<T>` είναι μία από τις **συλλογές (Collections)** που είναι ορισμένες στην Java.
 - Υπάρχουσες **δομές δεδομένων** που μας βοηθάνε στην αποθήκευση και **ανάκτηση** των **δεδομένων**.

Η ιεραρχία



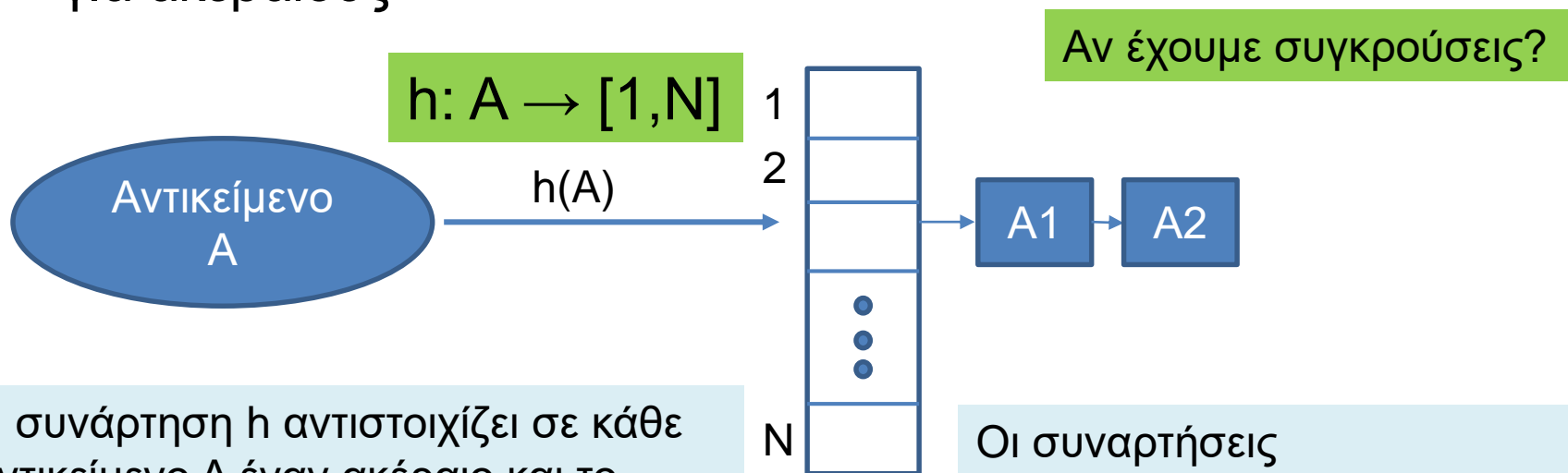
Αποθηκεύει δεδομένα σε **σειριακή** μορφή. Υπάρχει η έννοια της **διάταξης**. Καλό αν θέλουμε να **διατρέχουμε** τα δεδομένα συχνά και γρήγορα.

Αποθηκεύει δεδομένα σαν **σύνολο** χωρίς διάταξη. Καλό αν θέλουμε να βρίσκουμε γρήγορα αν ένα στοιχείο **ανήκει** στο σύνολο

Αποθηκεύει **(key,value)** ζεύγη. Παρόμοια δομή με το `HashSet` για την αποθήκευση των **κλειδιών**, αλλά τώρα κάθε κλειδί (key) **σχετίζεται** με μία **τιμή** (value).

Εισαγωγή μιας διαφάνειας σε Hashing

- Τα HashSet και HashMap χρησιμοποιούν **συναρτήσεις κατακερματισμού** (**hash functions**).
- Μια συνάρτηση κατακερματισμού μας επιτρέπει να δεικτοδοτούμε ένα πίνακα χρησιμοποιώντας αντικείμενα αντί για ακεραίους



Η συνάρτηση h αντιστοιχίζει σε κάθε αντικείμενο A έναν ακέραιο και το αποθηκεύουμε στην αντίστοιχη θέση. Άρα μπορούμε αμέσως να βρούμε αν το A ανήκει στο σύνολο μας

Οι συναρτήσεις κατακερματισμού είναι σχεδιασμένες έτσι ώστε να ελαχιστοποιείται η πιθανότητα σύγκρουσης

ArrayList ([JavaDocs link](#))

- Constructor

- `ArrayList<T> myList = new ArrayList<T>();`

- Μέθοδοι

- `add(T x)` : προσθέτει το στοιχείο `x` στο τέλος του πίνακα.
 - `add(int i, T x)` : προσθέτει το στοιχείο `x` στη θέση `i` και μετατοπίζει τα υπόλοιπα στοιχεία κατά μια θέση.
 - `remove(int i)` : αφαιρεί το στοιχείο στη θέση `i` και το επιστρέφει.
 - `remove(T x)` : αφαιρεί το στοιχείο
 - `set(int i, T x)` : θέτει στην θέση `i` την τιμή `x` αλλάζοντας την προηγούμενη
 - `get(int i)` : επιστρέφει το αντικείμενο τύπου `T` στη θέση `i`.
 - `contains(T x)` : boolean αν το στοιχείο `x` ανήκει στην λίστα ή όχι.
 - `size()` : ο αριθμός των στοιχείων του πίνακα.

- Διατρέχοντας τον πίνακα:

- `ArrayList<T> myList = new ArrayList<T>();`
 - `for(T x: myList) {...}`

HashSet ([JavaDocs link](#))

- Constructors

- `HashSet<T> mySet = new HashSet<T>();`

- Μέθοδοι

- `add(T x)` : προσθέτει το στοιχείο `x` αν δεν υπάρχει ήδη στο σύνολο.

- `remove(T x)` : αφαιρεί το στοιχείο `x`.

- `contains(T x)` : boolean αν το σύνολο περιέχει το στοιχείο `x` ή όχι.

- `size()` : ο αριθμός των στοιχείων στο σύνολο.

- `isEmpty()` : boolean αν έχει στοιχεία το σύνολο ή όχι.

- `Object[] toArray()` : επιστρέφει πίνακα με τα στοιχεία του συνόλου (επιστρέφει πίνακα από Objects – χρειάζεται downcasting μετά).

- Διατρέχοντας τα στοιχεία του συνόλου:

- `HashSet<T> mySet = new HashSet<T>();`

- `for (T x: mySet) {...}`

Παράδειγμα I

- Διαβάζουμε μια σειρά από Strings και θέλουμε να βρούμε όλα τα **μοναδικά** Strings
 - Π.χ. να φτιάξουμε το λεξικό ενός βιβλίου
- Πώς θα το υλοποιήσουμε αυτό?
 - Με ArrayList?
 - Πρέπει να κάνουμε πάρα πολλές συγκρίσεις
 - Με HashSet?
 - Η αναζήτηση ενός string γίνεται πολύ πιο γρήγορα.


```

import java.util.HashSet;
import java.util.Scanner;

public class HashSetExample
{
    public static void main(String[] args){
        HashSet<String> mySet = new HashSet<String>();
        Scanner input = new Scanner(System.in);

        while(input.hasNext()){
            String name = input.next();
            if (!mySet.contains(name)){
                mySet.add(name);
            }
        }

        for(String name: mySet){
            System.out.println(name);
        }

        Object[] array = mySet.toArray();
        for (int i = 0; i < array.length; i ++){
            String name = (String)array[i];
            System.out.println(name);
        }
    }
}

```

Δήλωση μιας μεταβλητής **HashSet** από Strings.

Τοποθετούμε στο HashSet μόνο τα Strings τα οποία δεν έχουμε ήδη δει (δεν είναι ήδη στο σύνολο)

Ένας τρόπος για να **διατρέξουμε** και να τυπώσουμε τα στοιχεία του HashSet

Ένας άλλος τρόπος για να διατρέξουμε το HashSet χρησιμοποιώντας την εντολή **toArray()**. Ο πίνακας είναι πίνακας από Objects, και πρέπει να κάνουμε **downcasting**

```
import java.util.HashSet;
import java.util.Scanner;

public class HashSetExample
{
    public static void main(String[] args){
        HashSet<String> mySet = new HashSet<String>();
        Scanner input = new Scanner(System.in);

        while(input.hasNext()){
            String name = input.next();
            mySet.add(name);
        }

        for(String name: mySet){
            System.out.println(name);
        }

        Object[] array = mySet.toArray();
        for (int i = 0; i < array.length; i++){
            String name = (String)array[i];
            System.out.println(name);
        }
    }
}
```

Επειδή το HashSet κρατάει μοναδικά αντικείμενα, δεν χρειάζεται να κάνουμε τον έλεγχο. Αν υπάρχει ήδη το String δεν θα το ξαναπροθέσει.

HashMap ([JavaDocs link](#))

- Αποθηκεύει ζευγάρια από τιμές και κλειδιά.
- Constructor
 - `HashMap<K, V> myMap = new HashMap<K, V> ();`
- Μέθοδοι
 - `put(K key, V value)` : προσθέτει το ζευγάρι (`key, value`) (δημιουργεί μία συσχέτιση)
 - `V get(K key)` : επιστρέφει την τιμή για το κλειδί `key`.
 - `remove(K key)` : αφαιρεί το ζευγάρι με κλειδί `key`.
 - `containsKey(K key)` : boolean αν το σύνολο περιέχει το κλειδί `key` ή όχι.
 - `containsValue(V value)` : boolean αν το σύνολο περιέχει την τιμή `value` ή όχι. (αργό)
 - `size()` : ο αριθμός των στοιχείων (κλειδιών) στο map.
 - `isEmpty()` : boolean αν έχει στοιχεία το map ή όχι.
 - `Set<K> keySet()` : επιστρέφει ένα `Set` με τα κλειδιά.
 - `Collection<V> values()` : επιστρέφει ένα `Collection` με τις τιμές
 - `Set<Map.Entry<K, V>> entrySet()` : επιστρέφει μία `Set` αναπαράσταση των key-value εγγραφών στο HashMap

Παράδειγμα II

- Διαβάζουμε μια σειρά από Strings και θέλουμε να βρούμε όλα τα μοναδικά Strings και να τους δώσουμε ένα μοναδικό id.
 - Π.χ. να δώσουμε αριθμούς σε μία λίστα με ονόματα
- Πώς θα το υλοποιήσουμε αυτό?
- Τι γίνεται αν θέλουμε να δημιουργήσουμε ένα αντικείμενο Person για κάθε μοναδικό όνομα?

```
import java.util.HashMap;
import java.util.Scanner;

public class HashMapExample1
{
```

```
    public static void main(String[] args) {
        HashMap<String,Integer> myMap = new HashMap<String,Integer>();
        Scanner input = new Scanner(System.in);
```

```
        int counter = 0;
        while(input.hasNext()) {
            String name = input.next();
            if (!myMap.containsKey(name)) {
                myMap.put(name, counter);
                counter++;
            }
        }
    }
```

Διατρέχοντας το HashMap

```
        for(String name: myMap.keySet()) {
            System.out.println(name + ":" + myMap.get(name));
        }
    }
```

Δήλωση μιας μεταβλητής **HashMap** που συσχετίζει **Strings** (κλειδιά) και **Integers** (τιμές) Για κάθε όνομα (String) το id (Integer)

Αν το όνομα δεν είναι ήδη στο HashMap τότε ανάθεσε στο όνομα αυτό τον επόμενο αύξοντα αριθμό και πρόσθεσε ένα νέο ζευγάρι (όνομα αριθμός) στο HashMap.

Διέτρεξε το σύνολο με τα κλειδιά (ονόματα) στο HashMap

Για κάθε κλειδί (όνομα) πάρε το id που αντιστοιχεί στο όνομα αυτό και τύπωσε το.

```
import java.util.HashMap;
import java.util.Scanner;

public class HashMapExample2
{
    public static void main(String[] args){
        HashMap<String,Person> myMap = new HashMap<String,Person>();
        Scanner input = new Scanner(System.in);

        int counter = 0;
        while(input.hasNext()){
            String name = input.next();
            if (!myMap.containsKey(name)){
                Person p = new Person(name,counter);
                myMap.put(name,p);
                counter++;
            }
        }

        for(String name: myMap.keySet()){
            System.out.println(myMap.get(name));
        }
    }
}
```

Δημιουργούμε ένα HashMap το οποίο σε κάθε διαφορετικό όνομα αντιστοιχεί ένα **αντικείμενο** Person.

Καλείται η toString της κλάσης Person

Παράδειγμα III

- Διαβάζουμε μια σειρά από Strings και θέλουμε να βρούμε όλα τα μοναδικά Strings και να τον αριθμό των εμφανίσεων τους στο κείμενο.
- Πώς θα το υλοποιήσουμε αυτό?

```
import java.util.HashMap;
import java.util.Scanner;

class HashMapExample3
{
    public static void main(String[] args){
        HashMap<String, Integer> myMap = new HashMap<String,Integer>();
        Scanner input = new Scanner(System.in);

        while(input.hasNext()){
            String name = input.next();
            if (!myMap.containsKey(name)){
                myMap.put(name,1);
            }else{
                myMap.put(name,myMap.get(name)+1);
            }
        }

        for(String name: myMap.keySet()){
            System.out.println(name+": "+myMap.get(name));
        }
    }
}
```


Iterators

- Ένα interface που μας δίνει τις λειτουργίες για να **διατρέχουμε** ένα Collection
 - Ιδιαίτερα χρήσιμοι αν θέλουμε να **αφαιρέσουμε** στοιχεία από ένα Collection.
- Μέθοδοι του **Iterator<T>**
 - **hasNext ()** : boolean αν ο iterator έχει φτάσει στο τέλος ή όχι.
 - **T next ()** : επιστρέφει την επόμενη τιμή (αναφορά όχι αντίγραφο)
 - **remove ()** : αφαιρεί το στοιχείο το οποίο επέστρεψε η τελευταία next()
- Μέθοδος του **Collection** :
 - **Iterator iterator ()** : επιστρέφει ένα iterator για μία συλλογή.
Π.χ.:
 - `HashSet<String> mySet = new HashSet<String> ();`
 - `Iterator<String> iter = mySet.iterator ();`

```
import java.util.HashSet;
import java.util.Scanner;

public class WrongIteratorExample
{
    public static void main(String[] args){
        HashSet<String> mySet = new HashSet<String>();
        Scanner input = new Scanner(System.in);

        while(input.hasNext()){
            if (!mySet.contains(name)){
                mySet.add(input.next());
            }
        }

        for (String s: mySet){
            if (s.length() <= 2){
                mySet.remove(s);
            }
        }

        for (String s:mySet){
            System.out.println(s);
        }
    }
}
```

Θέλω να αφαιρέσω από το σύνολο τα Strings με λιγότερους από 2 χαρακτήρες

Αν διατρέξουμε το set με την for-each εντολή θα πάρουμε (συνήθως) **λάθος**.

Δεν μπορούμε να αλλάζουμε το Collection ενώ το διατρέχουμε!

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Scanner;
```

```
public class IteratorExample
{
```

```
    public static void main(String[] args) {
        HashSet<String> mySet = new HashSet<String>();
        Scanner input = new Scanner(System.in);
```

```
        while(input.hasNext()) {
            if (!mySet.contains(name)) { mySet.add(input.next()); }
        }
```

```
        Iterator<String> it = mySet.iterator();
```

```
        while (it.hasNext()) {
            if (it.next().length() <= 2) {
                it.remove();
            }
        }
```

```
        it = mySet.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
```

```
    }
}
```

Θέλω να αφαιρέσω από το σύνολο τα Strings με λιγότερους από 2 χαρακτήρες

Ο Iterator μας επιτρέπει να διατρέχουμε την συλλογή και να διαγράφουμε στοιχεία.

Ξανα-διατρέχουμε τον πίνακα. Ο iterator πρέπει να ξανα-οριστεί για να ξεκινήσει από την αρχή του συνόλου.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Scanner;
```

```
class IteratorExample2
```

```
{
```

```
    public static void main(String[] args){
```

```
        HashMap<String, Integer> myMap = new HashMap<String,Integer>();
```

```
        Scanner input = new Scanner(System.in);
```

```
        while(input.hasNext()){
```

```
            String name = input.next();
```

```
            if (!myMap.containsKey(name)) {myMap.put(name,1);}
```

```
            else{ myMap.put(name,myMap.get(name)+1);}
```

```
        }
```

```
        Iterator<Map.Entry<String,Integer>> iter = myMap.entrySet().iterator();
```

```
        while(iter.hasNext()){
```

```
            if (iter.next().getValue() <=2){
```

```
                iter.remove();
```

```
            }
```

```
        }
```

```
        for(String key: myMap.keySet()){
```

```
            System.out.println(key + ":" + myMap.get(key));
```

```
        }
```

```
    }
```

```
}
```

Θέλω να αφαιρέσω από το σύνολο τα Strings με λιγότερες από 2 εμφανίσεις

Η `entrySet` επιστρέφει μια συλλογή από `Map.Entry` αντικείμενα (γι αυτό πρέπει να κάνουμε `import` το `Map`) τα οποία παραμετροποιούμε με τους τύπους που κρατά το `HashMap`

ListIterator<T>

- Ένας Iterator ειδικά για την συλλογή List
 - Κύριο **πλεονέκτημα** ότι επιτρέπει διάσχιση της λίστας προς τις **δύο κατευθύνσεις** και **αλλαγές** στη λίστα **ενώ την διατρέχουμε**.
- Επιπλέον μέθοδοι της **ListIterator**
 - **hasPrevious()** : boolean αν υπάρχουν κι άλλα στοιχεία πριν από αυτό στο οποίο είμαστε.
 - **T previous()** : επιστρέφει την προηγούμενη τιμή
 - **set(T)** : Θέτει την τιμή του στοιχείου που επέστρεψε η τελευταία next()
 - **add(T)** : Προσθέτει ένα στοιχείο στη λίστα αμέσως μετά από αυτό στο οποίο βρισκόμαστε
- Μέθοδος της **List** :
 - **ListIterator listIterator()** : επιστρέφει ένα iterator για μία συλλογή.

Κάθε a μετέτρεψε το σε b και πρόσθεσε μετά και ένα c

```
import java.util.*;

public class ListIteratorExample
{
    public static void main(String[] args){
        ArrayList<String> array = new ArrayList<String>();
        Scanner input = new Scanner(System.in);

        while(input.hasNext()){
            String name = input.next();
            array.add(name);
        }

        ListIterator<String> it = array.listIterator();
        while (it.hasNext()){
            if (it.next().equals("a")){
                it.set("b");
                it.add("c");
            }
        }
        it = array.listIterator();
        while (it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

```
import java.util.*;

public class ListIteratorExample
{
    public static void main(String[] args) {
        ArrayList<String> myList = new ArrayList<String>();
        Scanner input = new Scanner(System.in);

        while (input.hasNext()) {
            String name = input.next();
            myList.add(name);
        }

        myList.remove("a");

        for (String s: myList) {
            System.out.println(s);
        }
    }
}
```

Θέλω να αφαιρέσω από τις εμφανίσεις του String "a" από την λίστα μου

Η κλήση της **remove** θα αφαιρέσει μόνο την **πρώτη εμφάνιση** του "a"

Πως θα τις αφαιρέσουμε όλες?

Υπενθύμιση: η **remove** επιστρέφει boolean αν έγινε επιτυχώς αφαίρεση (αν άλλαξε δηλαδή η λίστα).

```
import java.util.*;
```

Θέλω να αφαιρέσω από τις εμφανίσεις του String "a" από την λίστα μου

```
public class ListIteratorExample
```

```
{
```

```
    public static void main(String[] args) {
```

```
        ArrayList<String> myList = new ArrayList<String>();
```

```
        Scanner input = new Scanner(System.in);
```

```
        while (input.hasNext()) {
```

```
            String name = input.next();
```

```
            myList.add(name);
```

```
        }
```

Καλεί την remove μέχρι να επιστρέψει false

```
        while (myList.remove("a"));
```

```
        for (String s: myList) {
```

```
            System.out.println(s);
```

```
        }
```

```
    }
```

```
}
```

Η υλοποίηση αυτή όμως **δεν είναι αποδοτική** γιατί κάθε φορά που καλούμε την remove διατρέχουμε την λίστα από την αρχή. Είναι καλύτερα να χρησιμοποιήσουμε ένα iterator.

Χρήση των συλλογών

- Οι τρεις συλλογές που περιγράψαμε είναι **πάρα πολύ χρήσιμες** για να κάνετε γρήγορα προγράμματα
 - Συνηθίσετε να τις χρησιμοποιείτε και μάθετε πότε βολεύει να χρησιμοποιείτε την κάθε δομή
- Το HashMap είναι ιδιαίτερα χρήσιμο γιατί μας επιτρέπει **πολύ γρήγορα** να κάνουμε **lookup**: να βρίσκουμε ένα **κλειδί** μέσα σε ένα σύνολο και την **συσχετιζόμενη τιμή**

Παραδείγματα

- Έχουμε ένα πρόγραμμα που διαχειρίζεται τους φοιτητές ενός τμήματος. Ποια συλλογή πρέπει να χρησιμοποιήσουμε αν θέλουμε να λύσουμε τα παρακάτω προβλήματα?
 1. Θέλουμε να μπορούμε να εκτυπώσουμε τις πληροφορίες για τους φοιτητές που παίρνουν ένα μάθημα.
 - **`ArrayList<Student> allStudents`**
 2. Θέλουμε να μπορούμε να τυπώσουμε τις πληροφορίες για ένα συγκεκριμένο φοιτητή (χρησιμοποιώντας το AM του φοιτητή)
 - **`HashMap<Integer, Student> allStudents`**
 3. Θέλουμε να ξέρουμε ποιοι φοιτητές έχουν ξαναπάρει το μάθημα και να μπορούμε να ανακτήσουμε αυτή την πληροφορία για κάποιο φοιτητή
 - **`HashSet<Integer> repeatStudents`**
 - **`HashSet<Student> repeatStudents`**

Αναζήτηση με AM

Αναζήτηση με αντικείμενο

Χρήση δομών

- **ArrayList**: όταν θέλουμε να **διατρέχουμε** τα αντικείμενα ή όταν θέλουμε διάταξη των αντικείμενων, και **δεν** θα χρειαστούμε **αναζήτηση** κάποιου αντικείμενου
 - Π.χ., μια κλάση Course περιέχει μια λίστα από αντικείμενα τύπου Students
 - Εφόσον μας ενδιαφέρει να τυπώνουμε **μόνο**.
- **HashSet**: όταν θέλουμε να έχουμε μια συλλογή από **μοναδικά** αντικείμενα και θέλουμε **γρήγορη αναζήτηση** για να μάθουμε αν κάποιο αντικείμενο ανήκει σε αυτή
 - Π.χ., να βρούμε αν ένας φοιτητής (AM) ανήκει στη λίστα των φοιτητών που ξαναπαίρνουν το μάθημα
 - Π.χ., να βρούμε τα μοναδικά ονόματα από μια λίστα με ονόματα με επαναλήψεις
- **HashMap**: **Ίδια** λειτουργικότητα με το **HashSet** αλλά μας επιτρέπει να **συσχετίσουμε** μια **τιμή** με κάθε στοιχείο του συνόλου
 - Π.χ. θέλω να ανακαλέσω γρήγορα τις πληροφορίες για ένα φοιτητή χρησιμοποιώντας το AM του
 - Το HashMap είναι πιο χρήσιμο απ' ό,τι ίσως θα περιμένατε

Περίπλοκες δομές

- Έχουμε μάθει τρεις βασικές δομές
 - `ArrayList`
 - `HashSet`
 - `HashMap`
- Μπορούμε να δημιουργήσουμε αντικείμενα που συνδιάζουν αυτές τις δομές
 - `HashMap<String, ArrayList<String>>`
 - `ArrayList<HashSet<String>>`
 - `HashMap<Integer, HashMap<String, String>>`

Παράδειγμα

- Θέλουμε για καθένα από τα μοναδικά Strings που διαβάζουμε να κρατάμε τις **θέσεις** στις οποίες εμφανίστηκαν.
 - Π.χ., αν έχουμε είσοδο “a b a c b a”, για το “a” θα τυπώσουμε τις θέσεις 0,2,5, για το “b” θα τυπώσουμε τις θέσεις 1,4 και για το “c” τη θέση 3.

```
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Scanner;

class HashMapArrayListExample
{
    public static void main(String[] args){
        HashMap<String,ArrayList<Integer>> myMap =
            new HashMap<String,ArrayList<Integer>>();
        Scanner input = new Scanner(System.in);

        int counter = 0;
        while(input.hasNext()){
            String name = input.next();
            if (!myMap.containsKey(name)){
                myMap.put(name,new ArrayList<Integer>());
            }
            myMap.get(name).add(counter);
            counter ++;
        }

        for(String name: myMap.keySet()){
            System.out.print(name + ":");
            for (Integer i:myMap.get(name)){
                System.out.print(" "+i);
            }
            System.out.println();
        }
    }
}
```

Παράδειγμα

- Στο πρόγραμμα της γραμματείας ενός πανεπιστημίου που κρατάει πληροφορία για τους φοιτητές, θέλω γρήγορα με το ΑΜ του φοιτητή να μπορώ να βρω το βαθμό για ένα μάθημα χρησιμοποιώντας τον κωδικό του μαθήματος. Τι δομή πρέπει να χρησιμοποιήσω?

Υλοποίηση

- Χρειαζόμαστε ένα **HashMap** με **κλειδί το AM** του φοιτητή ώστε να μπορούμε γρήγορα να βρούμε πληροφορίες για τον φοιτητή.
 - Τι τιμές θα κρατάει το HashMap?
- Θα πρέπει να κρατάει άλλο ένα **HashMap** το οποίο να έχει σαν **κλειδί τον κωδικό του μαθήματος** και σαν **τιμή τον βαθμό του φοιτητή**.

Ορισμός

```
HashMap<Integer, HashMap<Integer, double>> StudentCoursesGrades;
```

Χρήση

```
StudentCoursesGrades = new HashMap<Integer, HashMap<int, double>> ();  
StudentCoursesGrades.put(469, new HashMap<Integer, double>());  
StudentCoursesGrades.get(469).put(205, 9.5);  
StudentCoursesGrades.get(469).get(205);
```

Προσθέτει το βαθμό

Διαβάζει το βαθμό

Διαφορετική υλοποίηση

- Στο πρόγραμμα μου να έχω μια κλάση **Student** που κρατάει τις πληροφορίες για ένα φοιτητή και μία κλάση **StudentRecord** που κρατάει την καρτέλα του φοιτητή για το μάθημα. Πως αλλάζει η υλοποίηση?

Ορισμός

```
HashMap<Integer, Student> allStudents;
```

Ορισμός

```
class Student
{
    private int AM;
    private HashMap<Integer, StudentRecord> courses;
    ...

    public StudentRecord getCourseRecord(int CourseId) {
        return courses.get(courseId);
    }
}
```

Ορισμός

```
class StudentRecord
{
    private double grade;
    ...

    public double getGrade{
        return grade;
    }
}
```

Χρήση

```
allStudents.get(469).getCourseRecord(205).getGrade();
```

Ορισμός

```
HashMap<Integer, Student> allStudents;
```

Ορισμός

```
class Student
{
    private int AM;
    private HashMap<Integer, StudentRecord> courses;
    ...

    public HashMap<Integer, StudentRecord> getCourses () {
        return courses;
    }
}
```

Διαφορετική υλοποίηση
Μπορούμε να επιστρέψουμε
ένα HashMap

Ορισμός

```
class StudentRecord
{
    private double grade;
    ...

    public double getGrade{
        return grade;
    }
}
```

Χρήση

```
allStudents.get(469).getCourses().get(205).getGrade();
```

Χρονική πολυπλοκότητα

- Έχει τόσο μεγάλη σημασία τι δομή θα χρησιμοποιήσουμε? Όλες οι δομές μας δίνουν περίπου την ίδια λειτουργικότητα.
 - **ΝΑΙ!**
- Αν κάνουμε αναζήτηση για μια τιμή σε ένα **ArrayList** **πρέπει να διατρέξουμε τη λίστα** για να δούμε αν ένα στοιχείο ανήκει ή όχι στη λίστα.
 - Κατά μέσο όρο θα συγκρίνουμε με τα μισά στοιχεία της λίστας
 - Η χρονική πολυπλοκότητα είναι γραμμική ως προς τον αριθμό των στοιχείων
- Σε ένα **HashSet** ή **HashMap** αυτό γίνεται σε **χρόνο σχεδόν σταθερό** (ή λογαριθμικό ως προς τον αριθμό των στοιχείων)
 - Αν έχουμε πολλά στοιχεία, και κάνουμε πολλές αναζητήσεις αυτό κάνει διαφορά

```
import java.util.*;
```

```
class ArrayHashComparison
{
    public static void main(String[] args){
        ArrayList<Integer> array = new ArrayList<Integer>();
        for (int i =0; i < 100000; i ++){
            array.add(i);
        }
        HashSet<Integer> set = new HashSet<Integer>();
        for (int i =0; i < 100000; i ++){
            set.add(i);
        }
        ArrayList<Integer> randomNumbers = new ArrayList<Integer>();
        Random rand = new Random();
        for (int i = 0; i < 100000; i ++){
            randomNumbers.add(rand.nextInt(200000));
        }

        long startTime = System.currentTimeMillis();
        for (Integer x:randomNumbers){
            boolean b = array.contains(x);
        }
        long endTime = System.currentTimeMillis();
        long duration = (endTime - startTime);
        System.out.println("Array took "+ duration + " millisecs");

        startTime = System.currentTimeMillis();
        for (Integer x:randomNumbers){
            boolean b = set.contains(x);
        }
        endTime = System.currentTimeMillis();
        duration = (endTime - startTime);
        System.out.println("Set took "+duration + " millisecs");
    }
}
```

Με το ArrayList κάνουμε περίπου $100000 * 100000 / 2$ συγκρίσεις

Με το HashSet κάνουμε περίπου 100000 συγκρίσεις

14. ΓΡΑΦΙΚΑ ΠΕΡΙΒΑΛΛΟΝΤΑ (GUI)

Η βιβλιοθήκη SWING
Event-driven programming

Swing

- Τα **GUIs** (**Graphical User Interfaces**) είναι τα συνηθισμένα interfaces που χρησιμοποιούν παράθυρα, κουμπιά, menus, κλπ
- Η **Swing** είναι βιβλιοθήκη της Java για τον προγραμματισμό τέτοιων interfaces.
 - Η μετεξέλιξη του **AWT** (**Abstract Window Toolkit**) το οποίο ήταν το πρώτο αλλά όχι τόσο επιτυχημένο πακέτο της Java για GUI.
 - Τώρα έχει αντικατασταθεί από την βιβλιοθήκη JavaFX αλλά η Swing είναι πιο απλή για την εισαγωγή εννοιών.

Event driven programming

- Το Swing ακολουθεί το μοντέλο του **event-driven programming**
 - Υπάρχουν κάποια αντικείμενα που **πυροδοτούν συμβάντα** (firing an event)
 - Υπάρχουν κάποια άλλα αντικείμενα που είναι **ακροατές (listeners)** για συμβάντα.
 - Αν προκληθεί ένα συμβάν υπάρχουν ειδικοί **χειριστές** του συμβάντος (**event handlers**) – μέθοδοι που χειρίζονται ένα συμβάν
 - Το **συμβάν (event)** είναι κι αυτό ένα αντικείμενο το οποίο **μεταφέρει πληροφορία** μεταξύ του αντικειμένου που προκαλεί το συμβάν και του ακροατή.
- Σας θυμίζουν κάτι όλα αυτά?
 - Πολύ παρόμοιες αρχές υπάρχουν στην δημιουργία και τον χειρισμό **εξαιρέσεων**.

Swing

- Στην Swing βιβλιοθήκη ένα GUI αποτελείται από πολλά στοιχεία/συστατικά (**components**)
 - π.χ. παράθυρα, κουμπιά, μενού, κουτιά εισαγωγής κειμένου, κλπ.
- Τα components αυτά **πυροδοτούν συμβάντα**
 - Π.χ. το πάτημα ενός κουμπιού, η εισαγωγή κειμένου, η επιλογή σε ένα μενού, κλπ
- Τα συμβάντα αυτά τα χειρίζονται τα **αντικείμενα-ακροατές**, που έχουν ειδικές μεθόδους γι αυτά
 - Τι γίνεται όταν πατάμε ένα κουμπί, όταν κάνουμε μια επιλογή κλπ
- Όλο το πρόγραμμα κυλάει ως μια αλληλουχία από **συμβάντα** και τον **χειρισμό** τους από τους ακροατές.



JFrame

Το JFrame ορίζει ένα βασικό απλό παράθυρο.
Ο παρακάτω κώδικας δημιουργεί ένα παράθυρο

```
import javax.swing.JFrame;  
  
public class JFrameDemo  
{  
    public static final int WIDTH = 300;  
    public static final int HEIGHT = 200;  
  
    public static void main(String[] args)  
    {  
        JFrame firstWindow = new JFrame( );  
        firstWindow.setSize(WIDTH, HEIGHT);  
  
        firstWindow.setDefaultCloseOperation(  
            JFrame.EXIT_ON_CLOSE);  
  
        firstWindow.setVisible(true);  
    }  
}
```

Καθορίζει το μέγεθος
(πλάτος, ύψος) του
παραθύρου μετρημένο σε
pixels

Κάνει το παράθυρο ορατό

Καθορίζει τι κάνει το
παράθυρο όταν πατάμε
το κουμπί για κλείσιμο

JFrame

- Επιλογές για το `setDefaultCloseOperation`:
 - `EXIT_ON_CLOSE`: Καλεί την `System.exit()` και σταματάει το πρόγραμμα.
 - `DO_NOTHING_ON_CLOSE`: δεν κάνει τίποτα, ουσιαστικά δεν μας επιτρέπει να κλείσουμε το παράθυρο
 - `HIDE_ON_CLOSE`: Κρύβει το παράθυρο αλλά δεν σταματάει το πρόγραμμα.
- Άλλες μέθοδοι:
 - `add`: προσθέτει ένα συστατικό (component) στο παράθυρο (π.χ. ένα κουμπί)
 - `setTitle(String)`: δίνει ένα όνομα στο παράθυρο που δημιουργούμε.

ΕΤΙΚΕΤΕΣ

- Αφού έχουμε φτιάξει το βασικό παράθυρο μπορούμε πλέον να αρχίσουμε να **προσθέτουμε** συστατικά (**components**)
- Μπορούμε να προσθέσουμε ένα (σύντομο) κείμενο στο παράθυρο μας προσθέτοντας μια **ετικέτα (label)**
- **JLabel** class: μας επιτρέπει να δημιουργήσουμε μια ετικέτα με συγκεκριμένο κείμενο
 - `JLabel greeting = new JLabel("Hello World!");`
- Αφού δημιουργήσουμε την ετικέτα θα πρέπει να την **προσθέσουμε** μέσα στο παράθυρο μας.
 - Καλούμε την μέθοδο **add** της **JFrame**

Παράθυρο με ετικέτα

```
import javax.swing.JFrame;  
import javax.swing.JLabel;  
  
public class JLabelDemo  
{  
    public static final int WIDTH = 300;  
    public static final int HEIGHT = 200;  
  
    public static void main(String[] args)  
    {  
        JFrame firstWindow = new JFrame( );  
        firstWindow.setSize(WIDTH, HEIGHT);  
  
        firstWindow.setDefaultCloseOperation(  
            JFrame.EXIT_ON_CLOSE);  
        JLabel label = new JLabel("Hello World!");  
        firstWindow.add(label);  
  
        firstWindow.setVisible(true);  
    }  
}
```

Δημιουργία της ετικέτας με την κλάση
JLabel και προσθήκη στο παράθυρο

Κουμπιά

- Ένα άλλο component για ένα γραφικό περιβάλλον είναι τα **κουμπιά**.
- Δημιουργούμε κουμπιά με την κλάση **JButton**.
 - `JButton button = new JButton("click me");`
 - Το κείμενο στον constructor είναι αυτό που εμφανίζεται **πάνω** στο κουμπί.
- Για να ξέρουμε τι κάνει το κουμπί όταν πατηθεί θα πρέπει να συνδέσουμε το κουμπί με ένα **ακροατή**.
 - Ο ακροατής είναι ένα αντικείμενο μιας κλάσης που υλοποιεί το **interface ActionListener** η οποία έχει την μέθοδο
 - `actionPerformed(ActionEvent e)`: χειρίζεται ένα συμβάν
 - Αφού δημιουργήσουμε το αντικείμενο του ακροατή το **συνδέουμε (καταχωρούμε)** με το **κουμπί** χρησιμοποιώντας την μέθοδο της **JButton**:
 - `addActionListener(ActionListener)`

Παράθυρο με κουμπί

```
import javax.swing.JFrame;  
import javax.swing.JButton;  
  
public class ButtonDemo  
{  
    public static final int WIDTH = 300;  
    public static final int HEIGHT = 200;  
  
    public static void main(String[] args)  
    {  
        JFrame firstWindow = new JFrame( );  
        firstWindow.setSize(WIDTH, HEIGHT);  
  
        firstWindow.setDefaultCloseOperation(  
            JFrame.DO_NOTHING_ON_CLOSE);  
  
        JButton endButton = new JButton("Click to end program.");  
  
        EndingListener buttonEar = new EndingListener( );  
        endButton.addActionListener(buttonEar);  
  
        firstWindow.add(endButton);  
  
        firstWindow.setVisible(true);  
    }  
}
```

Δημιουργία κουμπιού με
την κλάση **JButton**

Δημιουργία και **καταχώριση**
του ακροατή στο κουμπί

Προσθήκη κουμπιού
στο παράθυρο

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class EndingListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}
```

Ένας ακροατής υλοποιεί το `interface ActionListener` και πρέπει να υλοποιεί την μέθοδο `actionPerformed(ActionEvent)`

Όταν πατάμε το κουμπί στο GUI καλείται η μέθοδος `actionPerformed` του ακροατή που έχουμε καταχωρίσει για το κουμπί

Η κλήση της `actionPerformed` από τον `ActionListener` γίνεται αυτόματα μέσω της βιβλιοθήκης Swing, δεν την κάνει ο προγραμματιστής

Η παράμετρος `ActionEvent` περιέχει πληροφορία σχετικά με το συμβάν που μπορεί να χρησιμοποιηθεί.

Πιο σωστός τρόπος να ορίσουμε το παράθυρο μας ως ένα τύπο παράθυρου που επεκτείνει την κλάση JFrame

```
import javax.swing.JFrame;  
import javax.swing.JButton;
```

```
public class FirstWindow extends JFrame
```

```
{  
    public static final int WIDTH = 300;  
    public static final int HEIGHT = 200;  
  
    public FirstWindow( )  
    {  
        super( );  
        setSize(WIDTH, HEIGHT);  
  
        setTitle("First Window Class");  
  
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);  
  
        JButton endButton = new JButton("Click to end program.");  
        endButton.addActionListener(new EndingListener( ));  
        add(endButton);  
    }  
}
```

Η δημιουργία του ActionListener γίνεται ως ανώνυμο αντικείμενο μιας και δεν θα το χρησιμοποιήσουμε ποτέ άμεσα

```
public class DemoButtonWindow
{
    public static void main(String[] args)
    {
        FirstWindow w = new FirstWindow( );
        w.setVisible(true);
    }
}
```

Εδώ δημιουργούμε το παράθυρο μας

Αυτό είναι και το σωστό σημείο να αποφασίσουμε αν το παράθυρο θα είναι visible ή όχι.

Πολλά συστατικά

- Αν θέλουμε να βάλουμε **πολλά** components μέσα στο παράθυρο μας τότε θα πρέπει να προσδιορίσουμε **που** θα τοποθετηθούν αλλιώς θα μπούνε το ένα πάνω στο άλλο.
- Αυτό γίνεται με την μέθοδο **setLayout** που καθορίζει την τοποθέτηση μέσα στο παράθυρο
 - Αυτό μπορεί να γίνει με διαφορετικούς τρόπους

FlowLayout

- Απλά τοποθετεί τα components το ένα μετά το άλλο από τα αριστερά προς τα δεξιά
- Καλούμε την μέθοδο

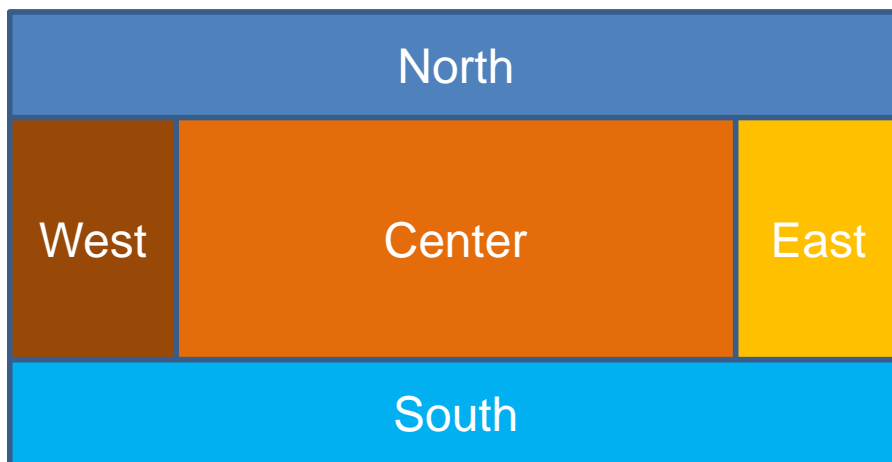
```
setLayout(new FlowLayout());
```

(Πρέπει να έχουμε κάνει `import java.awt.FlowLayout`)

- Μετά προσθέτουμε κανονικά τα components με την `add`.

BorderLayout

- Στην περίπτωση αυτή ο χώρος χωρίζεται σε πέντε περιοχές: North, South, East, West Center
- Καλούμε την μέθοδο
`setLayout(new BorderLayout());`
(Πρέπει να έχουμε κάνει `import java.awt.BorderLayout`)
- Μετά όταν προσθέτουμε τα components με την `add`, προσδιορίζουμε την περιοχή στην οποία θα προστεθούν.
 - Π.χ., `add(label, BorderLayout.CENTER)`



GridLayout

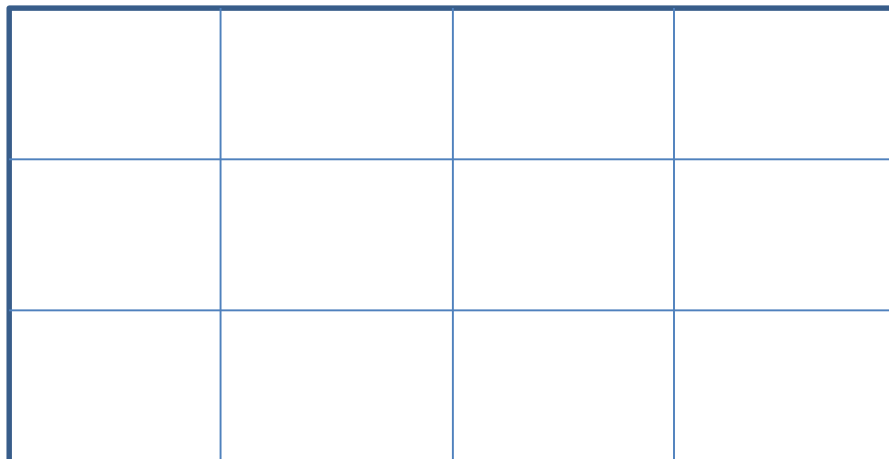
- Στην περίπτωση αυτή ορίζουμε ένα πλέγμα με n γραμμές και m στήλες και αυτό γεμίζει από τα αριστερά προς τα δεξιά και από πάνω προς τα κάτω
- Καλούμε την εντολή

```
setLayout(new GridLayout(n,m));
```

(Πρέπει να έχουμε κάνει `include java.awt.GridLayout`)

- Μετά προσθέτουμε κανονικά τα components με την `add`.

Grid 3x4



Παράδειγμα

- Δημιουργείστε ένα παράθυρο με τρία κουμπιά:
 - Το ένα κάνει το χρώμα του παραθύρου μπλε, το άλλο κόκκινο και το τρίτο κλείνει το παράθυρο.
 - Κώδικας: **MultiButtonWindow**

Η κλάση υλοποιεί τον ακροατή και την actionPerformed μεθοδο

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

```
public class MultiButtonWindow extends JFrame implements ActionListener
```

```
{
```

```
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;
```

```
    public MultiButtonWindow( )
```

```
    {
```

```
        super( "Multi-Color" );
        setSize(WIDTH, HEIGHT);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        setLayout(new FlowLayout());
```

```
        JLabel label = new JLabel("Pick A Color");
        add(label);
```

```
        JButton blueButton = new JButton("Blue");
        blueButton.addActionListener(this);
        add(blueButton);
```

```
        JButton redButton = new JButton("Red");
        redButton.addActionListener(this);
        add(redButton);
```

```
        JButton endButton = new JButton("Exit");
        endButton.addActionListener(this);
        add(endButton);
```

```
    }
```

Ορίζουμε τα χαρακτηριστικά του βασικού παραθύρου

Δημιουργούμε τα τρία κουμπιά και τα προσθέτουμε στο frame

Ο ακροατής των κουμπιών είναι το ίδιο το αντικείμενο (this)

Συνέχεια στην επόμενη

Συνέχεια από
την προηγούμενη

Η μέθοδος `actionPerformed` που καλείται όταν πατηθούν τα κουμπιά (μιας και το αντικείμενο είναι και ακροατής)

```
public void actionPerformed(ActionEvent e)
{
    String buttonType = e.getActionCommand( );

    switch (buttonType) {
        case "Blue":
            getContentPane().setBackground(Color.BLUE);
            break;
        case "Red":
            getContentPane().setBackground(Color.RED);
            break;
        case "Exit":
            System.exit(0);
    }
}
```

Το αποτέλεσμα του κάθε διαφορετικού κουμπιού.

Επιστρέφει το `actionCommand` String, το οποίο αν δεν το έχουμε αλλάξει είναι το όνομα του κουμπιού

Η `getContentPane` μας δίνει πρόσβαση στα χαρακτηριστικά του frame.
Η `setBackground` αλλάζει το χρώμα του frame

```
public static void main(String[] args)
{
    MultiButtonWindow w = new MultiButtonWindow();
    w.setVisible(true);
}
```

Δημιουργία του παραθύρου

Αξιοσημείωτα

- `public class MultiButtonWindow`
 - `extends JFrame`
 - `implements ActionListener`
- Μπορούμε να κάνουμε τον ακροατή να είναι το ίδιο το παράθυρο, αυτό θα αναλάβει να υλοποιήσει τη μέθοδο `actionPerformed`.
- Όταν καταχωρούμε τον ακροατή:
`blueButton.addActionListener(this);`
- `getContentPane().setBackground(Color.BLUE);`
 - Αλλάζει το background χρώμα του παραθύρου. Η κλάση `Color` μας δίνει τα χρώματα
- `String buttonType = e.getActionCommand();`
 - Με την εντολή αυτή παίρνουμε το `String` το οποίο δώσαμε σαν τίτλο στο κουμπί

actionCommand

- Ένα String πεδίο που κρατάει πληροφορία για το συμβάν
 - Αν δεν αλλάξουμε κάτι αυτό είναι το όνομα του κουμπιού
- Μπορούμε να διαβάσουμε το String με την εντολή `getActionCommand`.
- Μπορούμε να θέσουμε μια τιμή στο String με την εντολή `setActionCommand(String)`
- Π.χ.
`redButton.setActionCommand("RedButtonClick");`

Χρώματα

- Μπορούμε να ορίσουμε τα δικά μας χρώματα με την **RGB** σύμβαση
 - `Color myColor = new Color(200,100,4) ;`
 - Τα ορίσματα είναι οι RGB (**Red, Green, Blue**) τιμές

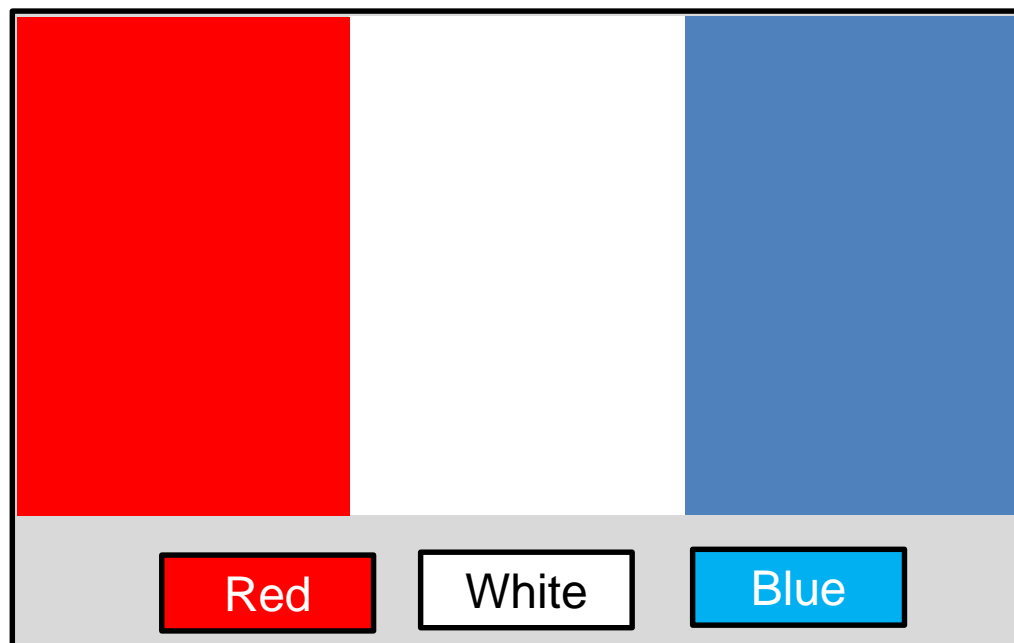
JPanel

- Το **panel** (τομέας) είναι ένας **container**
 - Μέσα σε ένα container μπορούμε να βάλουμε components και να ορίσουμε χειρισμό συμβάντων.
- Τα panels κατά μία έννοια ορίζουν ένα **παράθυρο μέσα στο παράθυρο**
 - Το panel έχει κι αυτό το δικό του layout και τοποθετούμε μέσα σε αυτό συστατικά.
 - Π.χ., ο παρακάτω κώδικας εκτελείται μέσα σε ένα JFrame.

```
setLayout(new BorderLayout());  
  
JPanel buttonPanel = new JPanel();  
buttonPanel.setLayout(new FlowLayout());  
  
JButton button1 = new JButton("one");  
buttonPanel.add(button1);  
  
JButton button2 = new JButton("two");  
buttonPanel.add(button2);  
  
add(buttonPanel, BorderLayout.SOUTH);
```

Παράδειγμα

- Θα δημιουργήσουμε ένα παράθυρο με τρία panels το κάθε panel θα παίρνει διαφορετικό χρώμα με ένα διαφορετικό κουμπί.



```
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

Η κλάση υλοποιεί τον ακροατή και την actionPerformed μεθοδο

```
public class PanelDemo extends JFrame implements ActionListener
```

```
{
```

```
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;
```

```
    private JPanel redPanel;
    private JPanel whitePanel;
    private JPanel bluePanel;
```

Δηλώνουμε τα τρία πάνελ με τα τρία χρώματα

```
public PanelDemo ( )
```

```
{
```

```
    super("Panel Demonstration");
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new BorderLayout ( ));
```

Ορίζουμε τα χαρακτηριστικά του βασικού παραθύρου

Συνέχεια στην επόμενη

Συνέχεια από
την προηγούμενη

Δημιουργούμε ένα μεγάλο πάνελ
που θα κρατάει τα τρία
χρωματιστά πάνελ

```
JPanel biggerPanel = new JPanel( );  
biggerPanel.setLayout(new GridLayout(1, 3));
```

```
redPanel = new JPanel( );  
redPanel.setBackground(Color.LIGHT_GRAY);  
biggerPanel.add(redPanel);
```

```
whitePanel = new JPanel( );  
whitePanel.setBackground(Color.LIGHT_GRAY);  
biggerPanel.add(whitePanel);
```

```
bluePanel = new JPanel( );  
bluePanel.setBackground(Color.LIGHT_GRAY);  
biggerPanel.add(bluePanel);
```

```
add(biggerPanel, BorderLayout.CENTER);
```

Δημιουργούμε τα
χρωματιστά
πάνελ και τα
προσθέτουμε
στο μεγάλο
πάνελ

Συνέχεια στην
επόμενη

Βάζουμε το μεγάλο πάνελ
στο κέντρο του παραθύρου

Συνέχεια από
την προηγούμενη

Δημιουργούμε ένα πάνελ που θα
κρατάει τα τρία κουμπιά

```
JPanel buttonPanel = new JPanel( );  
buttonPanel.setBackground(Color.LIGHT_GRAY);  
buttonPanel.setLayout(new FlowLayout( ));
```

```
JButton redButton = new JButton("Red");  
redButton.setBackground(Color.RED);  
redButton.addActionListener(this);  
buttonPanel.add(redButton);
```

```
JButton whiteButton = new JButton("White");  
whiteButton.setBackground(Color.WHITE);  
whiteButton.addActionListener(this);  
buttonPanel.add(whiteButton);
```

```
JButton blueButton = new JButton("Blue");  
blueButton.setBackground(Color.BLUE);  
blueButton.addActionListener(this);  
buttonPanel.add(blueButton);
```

```
add(buttonPanel, BorderLayout.SOUTH);
```

```
} // τέλος του constructor
```

Δημιουργούμε τα
τρία κουμπιά και
τα προσθέτουμε
στο πάνελ

Ο ακροατής των
κουμπιών είναι
το **ίδιο** το
αντικείμενο

Βάζουμε το πάνελ με τα κουμπιά
στον πάτο του παραθύρου

Συνέχεια στην
επόμενη

Συνέχεια από
την προηγούμενη

Η συνάρτηση `actionPerformed` που καλείται όταν πατηθούν τα κουμπιά (μιας και το αντικείμενο είναι και ακροατής)

```
public void actionPerformed(ActionEvent e)
{
    String buttonString = e.getActionCommand( );

    if (buttonString.equals("Red"))
        redPanel.setBackground(Color.RED);
    else if (buttonString.equals("White"))
        whitePanel.setBackground(Color.WHITE);
    else if (buttonString.equals("Blue"))
        bluePanel.setBackground(Color.BLUE);
    else
        System.out.println("Unexpected error.");
}

public static void main(String[] args)
{
    PanelDemo gui = new PanelDemo( );
    gui.setVisible(true);
}
}
```

Επιστρέφει το `actionCommand` String, το οποίο αν δεν το έχουμε αλλάξει είναι το όνομα του κουμπιού

Το αποτέλεσμα του κάθε διαφορετικού κουμπιού.

Δημιουργία του παραθύρου

Menu

- **Drop-down menus:**
 - **JMenuItem**: κρατάει μία από τις επιλογές του menu
 - **JMenu**: κρατάει όλα τα JMenuItemς
 - **JMenuBar**: κρατάει το Jmenu
 - **setJMenuBar (JMenu)** : θέτει το menu στην κορυφή του JFrame. Μπορούμε να χρησιμοποιήσουμε και τη γνωστή εντολή **add**.

Παράδειγμα

Δημιουργεί ένα drop-down menu

```
JMenu colorMenu = new JMenu("Add Colors");
```

```
JMenuItem redChoice = new JMenuItem("Red");  
redChoice.addActionListener(this);  
colorMenu.add(redChoice);
```

```
JMenuItem whiteChoice = new JMenuItem("White");  
whiteChoice.addActionListener(this);  
colorMenu.add(whiteChoice);
```

```
JMenuItem blueChoice = new JMenuItem("Blue");  
blueChoice.addActionListener(this);  
colorMenu.add(blueChoice);
```

```
JMenuBar bar = new JMenuBar();  
bar.add(colorMenu);  
setJMenuBar(bar);
```

Δημιουργεί τις επιλογές του μενού και τις προσθέτει στο μενού

Δημιουργεί ένα menu bar στην κορυφή του παραθύρου και προσθέτει το menu σε αυτό

Text Box

- Μπορούμε να δημιουργήσουμε ένα πεδίο κειμένου με την κλάση **JTextField**.
 - Το JTextField δημιουργεί ένα **text box** μίας γραμμής
 - **getText ()** : με την εντολή αυτή **διαβάζουμε** το κείμενο που δόθηκε σαν είσοδος στο text box.
 - **setText (String)** : με την εντολή αυτή **θέτουμε** το κείμενο στο text box.
- Για ένα πεδίο κειμένου μεγαλύτερο από μία γραμμή μπορούμε να χρησιμοποιήσουμε την κλάση **JTextArea**

Παράδειγμα

```
JTextField name = new JTextField(NUMBER_OF_CHAR);  
namePanel.add(name, BorderLayout.SOUTH);
```

```
JButton actionButton = new JButton("Click me");  
actionButton.addActionListener(this);  
buttonPanel.add(actionButton);
```

```
JButton clearButton = new JButton("Clear");  
clearButton.addActionListener(this);  
buttonPanel.add(clearButton);
```

```
public void actionPerformed(ActionEvent e)  
{  
    String actionCommand = e.getActionCommand( );  
  
    if (actionCommand.equals("Click me"))  
        name.setText("Hello " + name.getText( ));  
    else if (actionCommand.equals("Clear"))  
        name.setText("");  
    else  
        name.setText("Unexpected error.");  
}
```

Pop-up Windows

- Αν θέλουμε να δημιουργήσουμε παράθυρα διαλόγου μπορούμε να χρησιμοποιήσουμε την κλάση **JOptionPane**
 - Πετάει (pops up) ένα παράθυρο το οποίο μπορεί να μας ζητάει είσοδο, ή να ζητάει επιβεβαίωση.
 - Η δημιουργία και η διαχείριση των παραθύρων γίνεται με **στατικές μεθόδους**.

```

import javax.swing.JOptionPane;

public class PopUpDemo
{
    public static void main(String args[])
    {
        boolean done = false;
        while (!done){
            String classes =
                JOptionPane.showInputDialog("Enter number of classes");
            String students =
                JOptionPane.showInputDialog("Enter number of students");
            int totalStudents =
                Integer.parseInt(classes)*Integer.parseInt(students);

            JOptionPane.showMessageDialog(null,
                "Total number of students = "+totalStudents);

            int answer =
                JOptionPane.showConfirmDialog(null,
                    "Continue?",
                    "Confirm",
                    JOptionPane.YES_NO_OPTION);
            done = (answer == JOptionPane.NO_OPTION);
        }
        System.exit(0);
    }
}

```

Εμφανίζει ένα παράθυρο διαλόγου που ζητάει από τον χρήστη να δώσει είσοδο. Η είσοδος αποθηκεύεται στο String που επιστρέφεται

Το αντικείμενο (component) που είναι πατέρας του pop-up, null η default τιμή

Εμφανίζει ένα παράθυρο που τυπώνει ένα μήνυμα

Εμφανίζει ένα παράθυρο επιβεβαίωσης

Τύπος επιβεβαίωσης

Σταθερά για την επιλογή (YES_OPTION για ΝΑΙ)

Άλλοι τύποι επιβεβαίωσης:

- OK_CANCEL_OPTION
- YES_NO_CANCEL_OPTION

Η ερώτηση στο χρήστη

Τίτλος παραθύρου

Icons

- Μπορούμε να βάλουμε μέσα στο GUI μας και εικονίδια
- Παράδειγμα

Δημιουργεί ένα εικονίδιο από μία εικόνα

```
ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");  
JLabel dukeLabel = new JLabel("Mood check");  
dukeLabel.setIcon(dukeIcon);
```

Προσθέτει το εικονίδιο σε ένα label

```
ImageIcon happyIcon = new ImageIcon("smiley.gif");  
JButton happyButton = new JButton("Happy");  
happyButton.setIcon(happyIcon);
```

Προσθέτει το εικονίδιο σε ένα button

Ακροατές

- Στο πρόγραμμα μας ορίσαμε την κλάση που δημιουργεί το παράθυρο (**extends JFrame**) να είναι και ο ακροατής (**implements ActionListener**) των συμβάντων μέσα στο παράθυρο.
 - Αυτό είναι μια βολική λύση γιατί όλος ο κώδικας είναι στο **ίδιο** σημείο
 - Έχει το πρόβλημα ότι έχουμε **μία μόνο** μέθοδο **actionPerformed** στην οποία θα πρέπει να ξεχωρίσουμε όλες τις περιπτώσεις.
- Πιο βολικό να έχουμε ένα **διαφορετικό ActionListener** για κάθε διαφορετικό συμβάν
 - **Προβλήματα:**
 - Θα πρέπει να ορίσουμε **πολλαπλές κλάσεις** ακροατών σε πολλαπλά αρχεία
 - Θα πρέπει να περνάμε σαν παραμέτρους τα στοιχεία που θέλουμε να αλλάξουμε.

Ακροατές

- **Λύση:** Να ορίσουμε τους ακροατές που χρειάζεται το παράθυρο μας ως **εσωτερικές κλάσεις**

Εσωτερικές κλάσεις

- Μπορούμε να ορίσουμε μια κλάση μέσα στον ορισμό μιας άλλης κλάσης

```
class Shape
{
    private class Point
    {
        <Code for Point>
    }

    <Code for Shape>
}
```

Γιατί να το κάνουμε αυτό?

- Η κλάση `Point` μπορεί να είναι χρήσιμη **μόνο** για την `Shape`
- Μας επιτρέπει να ορίσουμε **άλλη** `Point` σε άλλο σημείο
- Η `Point` και η `Shape` έχουν η μία **πρόσβαση στα ιδιωτικά πεδία και μεθόδους** η μία της της άλλης

Ακροατές

- **Λύση:** Να ορίσουμε τους ακροατές που χρειάζεται το παράθυρο μας ως **εσωτερικές κλάσεις**
- **Πλεονεκτήματα:**
 - Οι κλάσεις είναι πλέον **τοπικές** στον κώδικα που τις καλεί, μπορούμε να επαναχρησιμοποιούμε τα ίδια ονόματα
 - Οι κλάσεις έχουν πρόσβαση σε **ιδιωτικά πεδία**

```
    JButton redButton = new JButton("Red");  
    redButton.setBackground(Color.RED);  
    redButton.addActionListener(new RedListener());  
    buttonPanel.add(redButton);
```

```
    JButton whiteButton = new JButton("White");  
    whiteButton.setBackground(Color.WHITE);  
    whiteButton.addActionListener(new WhiteListener());  
    buttonPanel.add(whiteButton);
```

```
    JButton blueButton = new JButton("Blue");  
    blueButton.setBackground(Color.BLUE);  
    blueButton.addActionListener(new BlueListener());  
    buttonPanel.add(blueButton);
```

Ορισμός των εσωτερικών κλάσεων-ακροατών (Κώδικας στο InnerListeners.java)

```
private class RedListener implements ActionListener{
    public void actionPerformed(ActionEvent e)
    {
        redPanel.setBackground(Color.RED);
    }
}

private class WhiteListener implements ActionListener{
    public void actionPerformed(ActionEvent e)
    {
        whitePanel.setBackground(Color.WHITE);
    }
}

private class BlueListener implements ActionListener{
    public void actionPerformed(ActionEvent e)
    {
        bluePanel.setBackground(Color.BLUE);
    }
}
```

Οι εσωτερικές κλάσεις έχουν πρόσβαση στα ιδιωτικά αντικείμενα πάνελ

Ανώνυμες κλάσεις

- Τα αντικείμενα-ακροατές είναι **ανώνυμα** αντικείμενα
 - `redButton.addActionListener(new RedListener());`
- Μπορούμε να κάνουμε τον κώδικα ακόμη πιο συνοπτικό ορίζοντας μια **ανώνυμη κλάση**
 - Ο ορισμός της κλάσης γίνεται εκεί που τον χρειαζόμαστε μόνο και **υλοποιεί ένα Interface**
 - Δεν συνίσταται αλλά μπορεί να το συναντήσετε σε κώδικα που δημιουργείται από IDEs

```
redButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e) {  
        redPanel.setBackground(Color.RED);  
    }  
});
```

Ο ορισμός της κλάσης
Χρησιμοποιούμε το **όνομα του interface**

Eclipse

- Η eclipse (αλλά και άλλα IDEs) μας δίνει πολλά έτοιμα εργαλεία για την δημιουργία GUIs
- Εγκαταστήσετε το plug-in **Windows Builder Pro**
- **Παράδειγμα:** Δημιουργήστε μια αριθμομηχανή.

Σχεδιασμός

Τα αποτελέσματα εμφανίζονται στην κορυφή τα πλήκτρα από κάτω

Textbox για να εκτυπώνει το αποτέλεσμα

Κουμπιά για καθένα από τα πλήκτρα

1	2	3	+
4	5	6	-
7	8	9	*
C	0	=	/

Χρειαζόμαστε ένα border layout για να βάλουμε το textbox στην κορυφή. Στο κέντρο θα βάλουμε τα κουμπιά. Βάζουμε ένα panel με grid layout

Εισαγωγή μίας διαφάνειας στο Eclipse

- Το Eclipse οργανώνει τον κώδικα σε **projects**.
- Οι **κλάσεις** στη συνέχεια προστίθενται μέσα στο project.
- Πρέπει να έχετε εγκαταστήσει το plugin **Windows Builder Pro**.
- Για να φτιάξετε ένα GUI
 - Αρχικά πρέπει να φτιάξετε ένα Java Project
 - Συνέχεια προσθέτετε στο project. Επιλέξετε **Other>WindowsBuilder>SWING>Application Window**.
- Στη συνέχεια θα έχετε ένα **μενού** από τα διάφορα components τα οποία μπορείτε να προσθέτετε στο στην εφαρμογή σας.
 - Μπορείτε να δουλεύετε είτε με το **Design** είτε με τον **Source** κώδικα



Package Explorer

- DM-ex5
 - test
 - src
 - (default package)
 - simple.java

- JRE System Library [JavaSE-1.7]

IndexFiles.java simple.java

Structure

Components

- btnNewButton_2 - "3"
- btnNewButton_3 - "+"
- btnNewButton_1 - "4"
- button - "5"
- btnNewButton_5 - "6"
- button_1 - "-"
- button_2 - "7"
- btnC - "8"
- button_4 - "9"
- button_5 - "*"
- btnC_1 - "C"
- button_6 - "0"
- button_7 - "="

Properties

Variable	button_6
Constructor	(Constructor properties)
Class	javax.swing.JButton
background	240,240,240
enabled	<input checked="" type="checkbox"/> true
font	Tahoma 11
foreground	0,0,0
horizontalAlign...	CENTER
icon	
mnemonic(char)	
selectedIcon	
text	0
toolTipText	
verticalAlignment	CENTER

Palette

- System
 - Selection
 - Marquee
 - Choose c...
 - Tab Order
- Containers
 - JPanel
 - JScrollPane
 - JSplitPane
 - JTabbedPane
 - JToolBar
 - JLayeredPane
 - JDesktopPane
 - InternalFrame
- Layouts
 - AbsoluteLayout
 - FlowLayout
 - BorderLayout
 - GridLayout
 - GridBagLayout
 - CardLayout
 - BoxLayout
 - SpringLayout
 - FormLayout
 - MigLayout
 - GroupLayout
- Struts & Springs
- Components
 - JLabel
 - JTextField
 - JComboBox
 - JButton
 - JCheckBox
 - JRadioButton
 - JToggleButton
 - JTextComponent
 - JFormattedTextField
 - JPasswordField
 - JTextPane
 - JEditorPane

JTextField
A lightweight component that allows the editing of a single line of text.

Source Design

Problems Javadoc Declaration

java.awt.event.ActionListener

The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked.

Since: 1.1
Author: Carl Quinn

Δημιουργία κώδικα

- Τα IDEs μας επιτρέπουν να διαχωρίζουμε το **design** από τον **κώδικα**
 - Το πλεονέκτημα είναι ότι έχουμε ένα **WYSIWYG** interface με το οποίο μπορούμε να σχεδιάσουμε το GUI
 - Το μειονέκτημα είναι ότι δημιουργείται πολύς κώδικας **αυτόματα** ο οποίος δεν είναι πάντα όπως τον θέλουμε.
- Ο **διαχωρισμός** του σχεδιαστικού κομματιού από τις πράξεις που εκτελούν είναι γενικά μια καλή προγραμματιστική πρακτική.

Δημιουργία κώδικα

- Η δημιουργία ενός κουμπιού δημιουργεί αυτό τον κώδικα

```
JButton button_6 = new JButton("0");  
panel.add(button_6);
```

- Αν πατήσουμε πάνω στο κουμπί (double-click) δημιουργείται ο ακροατής του κουμπιού αυτόματα ως μια **ανώνυμη κλάση**

```
JButton button_6 = new JButton("0");  
button_6.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
    }  
});  
panel.add(button_6);
```

Δημιουργία κώδικα

- Η δημιουργία ενός κουμπιού δημιουργεί αυτό τον κώδικα

```
JButton button_6 = new JButton("0");  
panel.add(button_6);
```

- Αν πατήσουμε πάνω στο κουμπί (double-click) δημιουργείται ο ακροατής του κουμπιού αυτόματα ως μια **ανώνυμη κλάση**
 - Εμείς συμπληρώνουμε τον κώδικα

```
JButton button_6 = new JButton("0");  
button_6.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        textField.setText(textField.getText()+"0");  
    }  
});  
panel.add(button_6);
```

15. ΕΠΙΣΚΟΠΗΣΗ

Θέματα που καλύψαμε

- Γενικές έννοιες αντικειμενοστραφούς προγραμματισμού
- Βασικά στοιχεία Java
- Κλάσεις και αντικείμενα
 - Πεδία, μέθοδοι, δημιουργοί, αναφορές
- Σύνθεση και συνάθροιση αντικειμένων
 - Πώς να φτιάχνουμε μεγαλύτερες κλάσεις με μικρότερα αντικείμενα - σχεδίαση
- Κληρονομικότητα, Πολυμορφισμός
- Συλλογές δεδομένων
- Εξαιρέσεις, I/O με αρχεία
- Γραφικά περιβάλλοντα

Αντικειμενοστραφής Προγραμματισμός

- Αν και το μάθημα έγινε σε Java, οι **βασικές αρχές** είναι οι ίδιες και για άλλες αντικειμενοστραφείς γλώσσες, και μπορείτε να μάθετε πολύ γρήγορα μια οποιαδήποτε **άλλη γλώσσα προγραμματισμού**
 - Μπορείτε να μάθετε **C#** σε μια βδομάδα
 - Η **C++** είναι λίγο πιο μπερδεμένη γιατί πρέπει να κάνετε μόνοι σας τη διαχείριση μνήμης αλλά με τις βασικές αρχές που ξέρετε μπορείτε να την μάθετε γρήγορα.

Εξετάσεις

- Οι εξετάσεις θα είναι με ανοιχτά βιβλία και σημειώσεις
- Οι ερωτήσεις θα είναι στο πνεύμα των εργαστηρίων και των ασκήσεων
 - Κατά κύριο λόγο θα είναι προγραμματιστικές, αλλά μπορεί να ζητηθεί να περιγράψετε ένα μηχανισμό, ή να εξηγήσετε γιατί συμβαίνει κάτι (κυρίως σε θέματα αναφορών)
- Καλή επιτυχία!