ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 1

Αλγόριθμος: Βήμα προς βήμα διαδικασία για την επίλυση κάποιου προβλήματος. Το πλήθος των βημάτων πρέπει να είναι πεπερασμένο.

Αλλιώς: Πεπερασμένη ακολουθία ενεργειών που περιγράφει την επίλυση κάποιου προβλήματος.

Πρόγραμμα: Ακριβής διατύπωση ενός αλγορίθμου σε μια γλώσσα προγραμματισμού.

Βήματα στην υπολογιστική επίλυση ενός προβλήματος:

1) Ανάλυση δεδομένων του προβλήματος
2) Μαθηματική διατύπωση του προβλήματος
3) Σχεδιασμός κατάλληλου αλγορίθμου
4) Ανάπτυξη προγράμματος (αλγόριθμος → σε γλώσσα προγραμματισμού)
5) Εκτέλεση προγράμματος για συγκεκριμένα δεδομένα
6) Ερμηνεία αποτελεσμάτων

Γλώσσες προγραμματισμού

i) Γλώσσες υψηλού επιπέδου (Java, C, C++, Python, Fortran, Pascal, κ.α.)
ii) Γλώσσες χαμηλού επιπέδου (γλώσσα μηχανής (Γ.Μ.), assembly)

Οι Η/Υ εκτελούν γλώσσα μηχανής (Γ.Μ.).

Γ.Μ.:
- διαφορετική για κάθε τύπο επεξεργαστή
- εντολές = αλληλουχίες από bits (0 και 1)

Assembly (λίγο ανώτερο επίπεδο από Γ.Μ.):
- διαφορετική για κάθε τύπο επεξεργαστή
- μνημονικά ονόματα αντί για bits

➢ Οι γλώσσες υψηλού επιπέδου (Γ.Υ.Ε.) μετατρέπονται σε γλώσσα μηχανής με κατάλληλα προγράμματα: τους **μεταγλωττιστές (compilers)**.

Γ.Υ.Ε. ⟶ ***compiler*** ⟶ **Γ.Μ.**

➢ Οι γλώσσες υψηλού επιπέδου είναι ανεξάρτητες από τον τύπο επεξεργαστή, άρα ο ίδιος κώδικας ενός προγράμματος σε κάποια γλώσσα είναι ίδιος για κάθε Η/Υ.

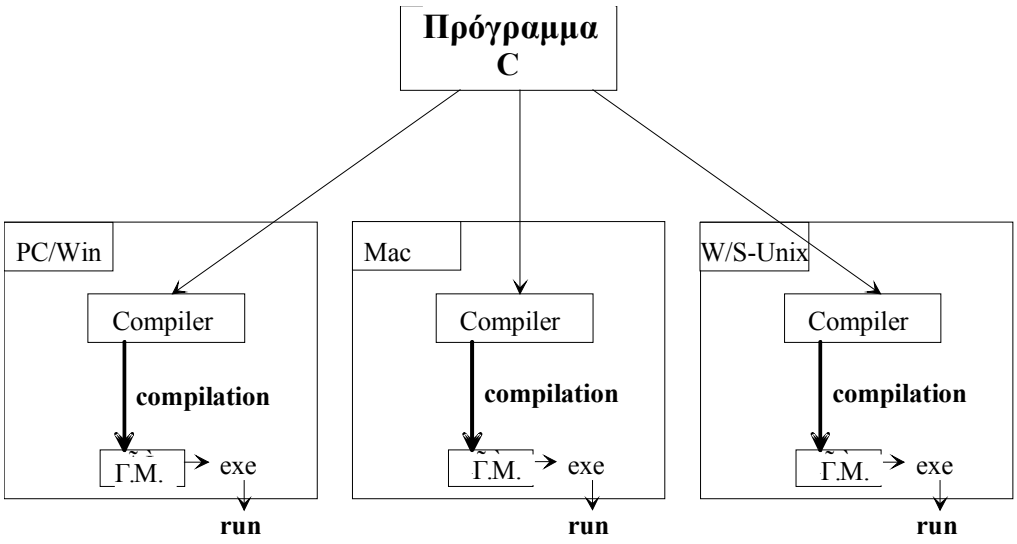
Όμως: Ο μεταγλωττιστής κάποιας τέτοιας γλώσσας είναι διαφορετικός για κάθε τύπο επεξεργαστή. Άρα, ένα πρόγραμμα σε κάποια γλώσσα υψηλού επιπέδου, μπορεί να είναι ίδιο για κάθε τύπο επεξεργαστή, όμως για να εκτελεσθεί χρειάζεται να μετατραπεί σε Γ.Μ. από τον αντίστοιχο μεταγλωττιστή της γλώσσας αυτής για τον συγκεκριμένο τύπο επεξεργαστή.

Σημείωση: Η μεταγλώττιση (compilation) ενός προγράμματος είναι συνήθως "χρονοβόρα" διαδικασία.

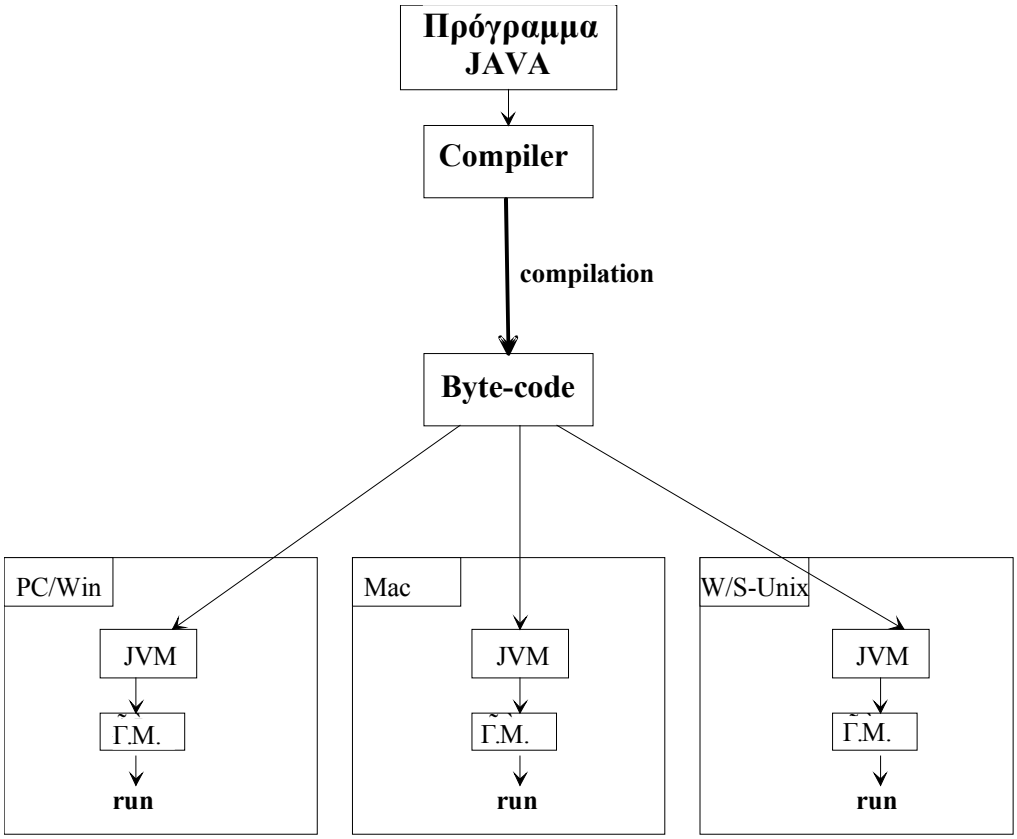
Το πλεονέκτημα της Java: Έχει **έναν** μεταγλωττιστή (compiler) για **όλους** τους τύπους επεξεργαστών. Ο compiler αυτός όμως **δεν** παράγει Γ.Μ., παράγει **byte-code**, μια «Γ.Μ.» για έναν εικονικό (ιδεατό) υπολογιστή. Αυτός ο εικονικός υπολογιστής (java virtual machine – JVM) είναι στην ουσία ένας διερμηνέας (interpreter) που μεταφράζει εντολή προς εντολή το byte-code σε Γ.Μ. του συγκεκριμένου επεξεργαστή στον οποίο τρέχει. Το JVM είναι διαφορετικό για κάθε επεξεργαστή, όμως είναι πολύ πιο απλό από έναν μεταγλωττιστή.

- Άρα, μπορεί η Java να προσθέτει ένα επιπλέον βήμα στην όλη διαδικασία, όμως πετυχαίνει τη μέγιστη **φορητότητα (portability)**, αφού χρειάζεται μόνο μία μεταγλώττιση (compilation) για να μπορεί να τρέξει σε οποιονδήποτε επεξεργαστή, ενώ η διερμηνεία είναι μια πολύ απλή διαδικασία.

Π.χ., για να τρέξει ένα πρόγραμμα C σε τρεις υπολογιστές που βασίζονται σε διαφορετικές πλατφόρμες (διαφορετικό επεξεργαστή ή/και λειτουργικό σύστημα), ακολουθείται η εξής διαδικασία:



Αντίστοιχα, για ένα πρόγραμμα Java η διαδικασία είναι η ακόλουθη:



Φαίνεται λοιπόν ότι, παρόλο που η Java προσθέτει ένα επιπλέον βήμα στην όλη διαδικασία, καταφέρνει να πραγματοποιεί *ένα μόνο compilation* του προγράμματος (που είναι η «επίπονη» διαδικασία, γι αυτό και φαίνεται στα σχήματα με μεγάλο βέλος), έναντι ενός compilation *για κάθε υπολογιστή* που απαιτεί η C. Αυτός είναι ο βασικός λόγος που η Java χρησιμοποιήθηκε και επικράτησε στο internet. Εάν κάποιος προσπαθούσε να υλοποιήσει τη διαδικασία της Java με κάποια άλλη γλώσσα προγραμματισμού (άρα έκανε το compilation αρχικά), θα έπρεπε **να ξέρει εκ των προτέρων** για τι είδους πλατφόρμα προορίζεται κάθε φορά το πρόγραμμα, κάτι που είναι εξαιρετικά ασύμφορο και πολλές φορές αδύνατο να πραγματοποιηθεί στο Internet.

Βασικά χαρακτηριστικά της Java:

1) Φορητότητα (portability): εκτέλεση μεταγλωττισμένου (compiled) κώδικα ανεξαρτήτως πλατφόρμας (→ Internet)
2) Αντικειμενοστραφής αλλά απλούστερη της C++
3) Μεγάλη βιβλιοθήκη έτοιμων κλάσεων
4) Χρησιμοποιεί στοιχεία της C
5) Ασφαλής
6) Κατανεμημένη (συντονισμός εκτέλεσης κώδικα διαφορετικών Η/Υ)

Ανάπτυξη εφαρμογής σε Java:

1) Κώδικας Java → κλάσεις → αρχεία. Π.χ., `Class1.java`

2) Compiler (μεταγλωττιστής)

`javac Class1.java` → `Class1.class` (byte-code)

3) Interpreter (διερμηνέας) (JVM)

`java Class1` → εκτέλεση του αρχείου Class1.class (αφού γίνει αυτόματα η διερμηνεία του από byte-code σε Γ.Μ.)

Μια εφαρμογή Java μπορεί να είναι είτε πρόγραμμα (οπότε τρέχει καλώντας τον JVM) είτε **applet**, δηλαδή ειδικό πρόγραμμα για το internet. Στην περίπτωση που μια εφαρμογή είναι applet, τρέχει είτε μέσω ενός web browser (π.χ. Chrome, Firefox, Opera, Safari, IE, κτλ.) είτε μέσω του appletviewer.

<u>Παράδειγμα μιας απλής εφαρμογής:</u> (Μία μόνο κλάση)

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

→ save: HelloWorld.java *(αποθήκευση του κώδικα σε αρχείο)*

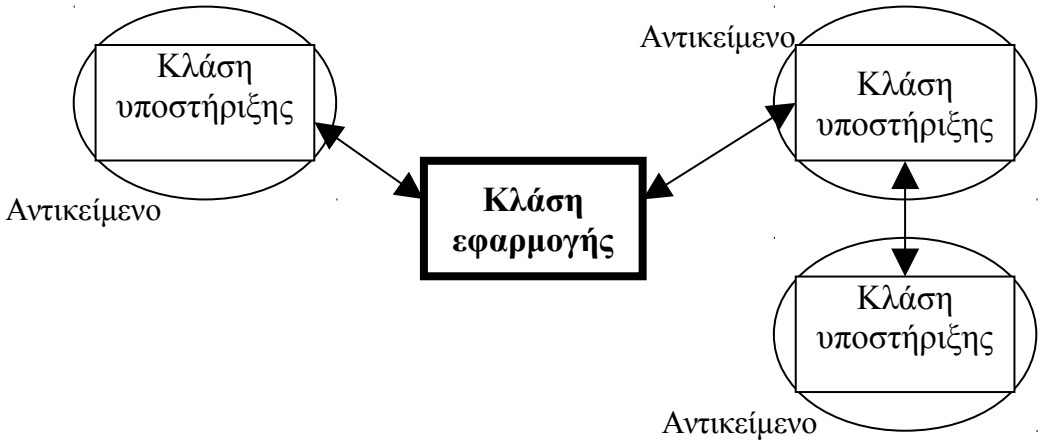
→ javac HelloWorld.java

→ java HelloWorld

Σε αυτό το σημείο το πρόγραμμα θα εκτελεσθεί και θα εμφανίσει στην οθόνη το μήνυμα:

Hello world!

Γενική μορφή ενός προγράμματος Java:



Ένα πρόγραμμα Java έχει απαραίτητα μία (και μόνο μία) *κλάση εφαρμογής* (η οποία περιέχει τη μέθοδο `main`) και *εάν* περιέχει και άλλες κλάσεις, τότε αυτές λέγονται *κλάσεις υποστήριξης*.

(Περισσότερα για τη γενική δομή ενός προγράμματος θα αναφερθούν σε επόμενες ενότητες, με την εισαγωγή στον αντικειμενοστραφή τρόπο προγραμματισμού).

Σφάλματα προγραμματισμού

α) Συντακτικό σφάλμα (syntax error) → κατά το compilation

β) Σφάλμα κατά την εκτέλεση (run-time error) → κατά τη διερμηνεία από το JVM (δηλ., περνάει το compilation αλλά η εκτέλεση του προγράμματος σταματάει.

γ) Λογικό σφάλμα (logic error / bug) → το πρόγραμμα εκτελείται αλλά το αποτέλεσμα είναι λάθος

- Δυσκολία εντοπισμού σφαλμάτων (συνήθως): **γ > β > α**
- Διαδικασία εντοπισμού και επίλυσης σφαλμάτων: **debugging**
- Διαδικασία ελέγχου κάποιων συγκεκριμένων σφαλμάτων (εξαιρέσεων): χειρισμός εξαιρέσεων (exceptions handling)

Μεταβλητές (variables)

Χρησιμοποιούνται για την αποθήκευση (φύλαξη) δεδομένων. Το στοιχείο το οποίο περιέχει μια μεταβλητή λέγεται **τιμή**.

Μια μεταβλητή έχει:
- ορατότητα (θα το δούμε αργότερα...)
- τύπο (type)
- όνομα (identifier – αναγνωριστικό)
- τιμή (value)

Η δημιουργία μιας μεταβλητής σε ένα πρόγραμμα γίνεται με τη δήλωσή της. Η σύνταξη μιας τέτοιας δήλωσης είναι:

```
<τύπος> <όνομα> [=τιμή];
```

(προς το παρόν παραλείπουμε την *ορατότητα*)

Στη διατύπωση ενός ορισμού σύνταξης μιας εντολής, το < ... > σημαίνει <u>υποχρεωτικό</u> μέρος της εντολής, ενώ το [...] σημαίνει <u>προαιρετικό</u>.

π.χ.,
```
int a = 5;
double b;
char answer;
```

- Άλλος τρόπος δήλωσης πολλών μεταβλητών μαζί (του ίδιου τύπου):

```
<τύπος> <μεταβλητή1>, <μεταβλητή2>, ... ;
```

π.χ., `int a1, a2, a3 = 5, a4;`

Τύποι (types) → πρωτογενείς (primitive types)
→ String (αλφαριθμητικά ή συμβολοσειρές)
→ κλάσεις → σύνθετη δομή σχεδιασμένη από τον προγραμματιστή

<u>Βασικοί κανόνες σύνταξης</u>

- ***Δεσμευμένες λέξεις (reserved):*** class, public, main, int, double, char, κτλ.
- ***Αναγνωριστικά (identifiers):*** ονόματα πραγμάτων (μεταβλητών, κλάσεων, μεθόδων, κτλ.)
  - *ξεκινάνε με:* γράμμα, _, $
  - *περιέχουν:* γράμματα, αριθμούς, _, $ (όχι κενά, τελείες, *, και λοιπά σύμβολα)
  - απεριόριστο πλήθος χαρακτήρων (συνήθως 1-15)
  - χωρίς κενά
  - υπάρχει διάκριση μεταξύ κεφαλαίων και μικρών

Συνήθως (κατά σύμβαση):
- ονόματα κλάσεων → ξεκινούν με Κεφαλαίο
- υπόλοιπα ονόματα (μεταβλητών, μεθόδων, κτλ) → ξεκινούν με μικρό
- σταθερές → όλα ΚΕΦΑΛΑΙΑ
- ονόματα γενικά: ξεκινούν με γράμμα και όχι με _ ή $ (αν και επιτρέπεται)
- ξεχωρίζουμε λέξεις ενός ονόματος με Κεφαλαίο γράμμα και όχι με _
  π.χ., `numberOfBaskets` αντί για `number_of_baskets`

Π.χ. κάποια ονόματα μεταβλητών:

`my.class`: λάθος (περιέχει τελεία)
`public`: λάθος (δεσμευμένη λέξη)
`7eleven`: λάθος (ξεκινάει με αριθμό)
`Var1`: σωστό, αλλά δε συνηθίζεται (μεταβλητή που ξεκινάει με Κεφαλαίο)

Πρωτογενείς τύποι μεταβλητών (primitive types)

byte	$-128 \leq$ ακέραιος ≤ 127
short	$-32768 \leq$ ακέραιος ≤ 32767
int	$-2^{31} \leq$ ακέραιος $\leq 2^{31}-1$ (δις)
long	$-2^{63} \leq$ ακέραιος $\leq 2^{63}-1$ (πεντάκις)
float	δεκαδικός, με εύρος: $\pm 10^{38}$ και ακρίβεια: $\pm 10^{-46}$
double	δεκαδικός, με εύρος: $\pm 10^{308}$ και ακρίβεια: $\pm 10^{-324}$
char	χαρακτήρας Unicode
boolean	true ή false

Θα χρησιμοποιούμε `int` για ακέραιους και `double` για πραγματικούς.

Παραδείγματα:

```
char symbol;
symbol = 'A';
System.out.println(symbol);
```

- Το σύμβολο "=" δεν είναι το μαθηματικό "=". Λέγεται **τελεστής εκχώρησης (assignment operator)**. Πραγματοποιεί *μεταβίβαση της τιμής στα δεξιά του στη μεταβλητή στα αριστερά του*.
  [Περισσότερες λεπτομέρειες θα αναφερθούν στην επόμενη ενότητα]

Άλλα παραδείγματα:

```
int a = 10;
boolean b = true;
double c = 4.2;
```

αλλά και:

```
a = a + 5;
```

```
int x = 1;
int y = 2;
x = y;   (το x γίνεται 2)
```

Τα δύο τελευταία παραδείγματα, αν και από μαθηματικής άποψης μπορούν να θεωρηθούν λάθος, σε μια γλώσσα προγραμματισμού είναι σωστά. Στο πρώτο, απλά αλλάζει η τιμή του `a` και από 10 γίνεται 15. Στο δεύτερο, η μεταβλητή `x` παίρνει την τιμή της μεταβλητής `y`, δηλαδή γίνεται ίση με 2.

- Για αλφαριθμητικά (αλληλουχία χαρακτήρων): Τύπος `String` (είναι κλάση – δεν είναι πρωτογενής τύπος μεταβλητών)

```
String s = "hello";
```

- Η τιμή σε μια μεταβλητή τύπου χαρακτήρα (`char`) δίνεται μέσα σε μονά εισαγωγικά (`'…'`) και είναι *ένας και μόνο ένας* χαρακτήρας.

- Η τιμή σε μια μεταβλητή τύπου `String` (αλφαριθμητικό) δίνεται μέσα σε διπλά εισαγωγικά (`"…"`) και μπορεί να είναι *από ένας μέχρι πολλοί χαρακτήρες* που μπορεί να έχουν και κενά μεταξύ τους *(περισσότερα σε επόμενο μάθημα)*.

- Η τιμή μιας μεταβλητής `boolean` μπορεί να είναι αποκλειστικά `true` ή `false` και δίνεται *χωρίς εισαγωγικά*. Συνήθως η τιμή μιας τέτοιας μεταβλητής προκύπτει από το αποτέλεσμα κάποιας λογικής πράξης, όπως θα δούμε σε επόμενο μάθημα.

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 2

Έξοδος στην οθόνη:

Η έξοδος στην οθόνη γίνεται με τις εντολές:

```
System.out.print("κείμενο");
```
ή
```
System.out.println("κείμενο");
```
για ταυτόχρονη αλλαγή γραμμής μετά την εκτύπωση.

Υπάρχει φυσικά η δυνατότητα εμφάνισης της τρέχουσας τιμής κάποιας μεταβλητής, με την εντολή:

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

(χωρίς εισαγωγικά αυτή τη φορά), καθώς και ο συνδυασμός κειμένου και τιμών μεταβλητών, με χρήση του τελεστή +:

```
System.out.println("κείμενο " + μεταβλητή);
```

Κάθε συμβολοσειρά (αλφαριθμητικό) είναι αντικείμενο της κλάσης `String` και όχι μια μεταβλητή πρωτογενούς τύπου (όπως π.χ. η `double`, η `int`, κτλ.).

Δημιουργία συμβολοσειρών (String): `String str = "Hello";`

Τελεστής συνένωσης (δημιουργία νέων String):

```
String s1 = "Hello ";
String s2 = "there";
String s3 = s1 + s2;
System.out.println(s3);   // → Hello there
```

- *«Πρόσθεση» (συνένωση) String με άλλους τύπους δεδομένων:*

```
System.out.println(4+7);  // → Τυπώνει: 11
System.out.println("Το άθροισμα είναι: " + 4 + 7);
     // → Τυπώνει: Το άθροισμα είναι: 47
```

Για να τυπωθεί το επιθυμητό 11, θα πρέπει να χρησιμοποιήσουμε παρένθεση στην πρόσθεση 4+7, ως εξής:

```
System.out.println("Το άθροισμα είναι: " + (4 + 7));
     // → Τυπώνει: Το άθροισμα είναι: 11
```

(Περισσότερα για την κλάση `String` θα αναφερθούν σε επόμενη ενότητα)

Είσοδος από το πληκτρολόγιο:

1) Προσθήκη απαραίτητης βιβλιοθήκης στην κλάση μας:

```
import java.util.*;
```
στην αρχή της κλάσης

2) Δημιουργία αντικειμένου της κλάσης Scanner (με όνομα π.χ. input):

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

3) Ερώτηση προς τον χρήστη για να ζητήσουμε την είσοδο:

```
System.out.println("……");
```

4) Ανάγνωση δεδομένων από το πληκτρολόγιο:

```
int n = input.nextInt();          // για ακέραιους
```
ή
```
double x = input.nextDouble();    // για πραγματικούς
```
ή
```
String s = input.nextLine();      // για συμβολοσειρές
```

Παράδειγμα εισόδου / εξόδου

Το παρακάτω πρόγραμμα Java αποτελείται από μία μόνο κλάση: την κλάση εφαρμογής (περιέχει τη μέθοδο `main`). Δέχεται δύο ακέραιους αριθμούς από τον χρήστη και εκτυπώνει στην οθόνη το άθροισμά τους.

```
/* A simple java input/output example.
   October 16, 2008 */

import java.util.*;      // "βιβλιοθήκη" που περιέχει την κλάση
                         // Scanner για είσοδο δεδομένων
public class Addition
{
   public static void main(String [] args)
   {
      System.out.println("Δώσε δύο ακέραιους αριθμούς:");
      int n1, n2;
      Scanner input = new Scanner(System.in);
      n1 = input.nextInt();
      n2 = input.nextInt();

      System.out.println("Το άθροισμα των δύο ακέραιων είναι: " + (n1+n2));

   }  // end of method

}  // end of class
```

Το κείμενο ανάμεσα στα σύμβολα `/*` και `*/` είναι *σχόλιο* και δεν αποτελεί μέρος του κώδικα της Java. Επίσης, το ίδιο συμβαίνει για το κείμενο που ακολουθεί το σύμβολο `//` σε μία γραμμή του προγράμματος (*σχόλιο γραμμής*).

- Σημείωση: Για την έξοδο στην οθόνη *δεν* χρειάζεται να προσθέσουμε κάποιο `import` στο πρόγραμμά μας. Ό,τι απαιτείται για την έξοδο εμπεριέχεται αυτόματα από τη Java. Αντίθετα, για την είσοδο, χρειάζεται να συμπεριλάβουμε (με το `import`) τη «βιβλιοθήκη» (πακέτο) java.util στο πρόγραμμά μας. *Περισσότερα για αυτή τη διαδικασία θα αναφερθούν σε επόμενη ενότητα.*

<u>Για να εκτελεσθεί το πρόγραμμά μας:</u>

i) Το αποθηκεύουμε σε αρχείο: **Addition.java**
ii) Το κάνουμε compile: **javac Addition.java** → Addition.class (bytecode)
iii) Εκτελούμε το αρχείο bytecode με τον interpreter: java Addition

Κατά την εκτέλεσή του το πρόγραμμα θα ζητήσει την είσοδο δύο ακεραίων από το πληκτρολόγιο και κατόπιν θα εκτυπώσει στην οθόνη το άθροισμά τους.

<u>Τελεστής εκχώρησης ή ανάθεσης (=)</u>

```
<μεταβλητή> = <τιμή>;
```
ή
```
<μεταβλητή> = <έκφραση>;
```

Έκφραση: εντολή ή πράξη που το αποτέλεσμά της είναι κάποια τιμή.

- Υπολογίζεται η έκφραση δεξιά τού = και η τιμή της αποθηκεύεται στη μεταβλητή που βρίσκεται αριστερά τού =
- Η προηγούμενη τιμή της μεταβλητής χάνεται.

<u>Ειδικοί τελεστές εκχώρησης (ή ανάθεσης)</u>

Υπάρχουν και κάποιοι ειδικοί τελεστές εκχώρησης:

```
a += x; → a = a + x;
a -= x; → a = a - x;
a *= x; → a = a * x;
a /= x; → a = a / x;

a++; }
     } → a = a + 1;
++a; }

a--; }
     } → a = a - 1;
--a; }
```

Διαφορά μεταξύ a++ και ++a:

Το αποτέλεσμα των a++ και ++a είναι το ίδιο, με τη διαφορά ότι όταν βρίσκονται μέσα σε μία έκφραση, στην περίπτωση του a++ *πρώτα* γίνεται η πράξη (χρησιμοποιώντας την αρχική τιμή του a) και *μετά* αυξάνεται το a κατά 1, ενώ στην περίπτωση του ++a *πρώτα* γίνεται η αύξηση της τιμής του a κατά 1 και *μετά* γίνεται η πράξη.

π.χ.
```
int a, b, c;
a=2;
a+=6;     → a=8
a-=3;     → a=5
a*=2;     → a=10
b = 2 * a++;  → a=11, b=20
c = 2 * ++a;  → a=12, b=24
```

Ειδική περίπτωση:

- Τι συμβαίνει όταν στην ίδια έκφραση υπάρχει πάνω από μία φορά ο τελεστής ++ (ή --) στην ίδια μεταβλητή (ή αν η μεταβλητή με τον εν λόγω τελεστή επανεμφανίζεται απλά μέσα στην έκφραση):

π.χ., 1η περίπτωση:
```
int a = 5;
int x = (a++) + (++a);    // → x = 5 + 7 = 12,  a = 7
```

2η περίπτωση:
```
int a = 5;
int x = (++a) + (a++);    // → x = 6 + 6 = 12,  a = 7
```

Δηλαδή, στη περίπτωση του `++a` η μεταβλητή πρώτα αυξάνεται κατά 1 και μετά παίρνει μέρος στην έκφραση, ενώ στην περίπτωση του `a++` παίρνει μέρος στην έκφραση με την αρχική της τιμή και αμέσως μετά αυξάνεται κατά 1, *οπότε αν υπάρχει και σε άλλο σημείο της έκφρασης, παίρνει μέρος με τη νέα της πλέον τιμή*.

Συμβατότητες εκχώρησης και αυτόματη μετατροπή τύπου

```
double doubleVar;
doubleVar = 7;   // ok: το 7 που είναι int μετατρέπεται σε double
```

Θα ήταν το ίδιο αν γράφαμε:
```
doubleVar = 7.0;
```

Επίσης:

```
int intVar;
intVar = 7;
double doubleVar;
doubleVar = intVar;  // ok: το ίδιο με πριν αλλά με μεταβλητή int
```

Γενικά, μια τιμή οποιουδήποτε τύπου μπορεί να εκχωρηθεί σε μια μεταβλητή οποιουδήποτε τύπου που βρίσκεται δεξιά από αυτόν στη λίστα:

`byte` → `short` → `int` → `long` → `float` → `double`

δηλαδή, μια τιμή τύπου `byte` μπορεί να μπει σε μια μεταβλητή τύπου `int`, ενώ μια τιμή τύπου `float` δεν μπορεί να μπει σε μια μεταβλητή τύπου `short`. κτλ.

(Μια τιμή τύπου `char` εκχωρείται σε μεταβλητή από `int` και πάνω)

Αριθμητικοί τελεστές και αυτόματη μετατροπή τύπου

Πράξεις: +, -, *, /, % (υπόλοιπο διαίρεσης)

- Το αποτέλεσμα μιας πράξης μεταξύ 2 μεταβλητών του ίδιου τύπου, είναι προφανώς του ίδιου τύπου με αυτόν των μεταβλητών
- Όταν 2 μεταβλητές είναι διαφορετικού τύπου, το αποτέλεσμα είναι του πιο «σύνθετου» τύπου (δηλ., του τύπου με τη μεγαλύτερη ακρίβεια):

`byte` → `short` → `int` → `long` → `float` → `double`

- Οι μεγαλύτερες εκφράσεις (πράξεις μεταξύ περισσότερων από 2 μεταβλητών) μπορούν πάντα να αναλυθούν σε βήματα με πράξεις 2 μεταβλητών, άρα το αποτέλεσμά τους είναι του πιο «σύνθετου» τύπου μεταξύ των τύπων όλων των μεταβλητών της έκφρασης.

π.χ., 3 * 4 → 12, 3 % 4 → 3, 11 % 3 → 2, 3 / 4 → 0 (γιατί είναι ακέραιοι!), 3.0 * 4 → 12.0

Μετατροπή τύπου (type casting)

Στις προηγούμενες περιπτώσεις η μετατροπή τύπου γινόταν αυτόματα. Όμως:

```
double distance;
distance = 9.0;
int points;
points = distance;   → ΣΦΑΛΜΑ!
```

Το σωστό θα ήταν:

```
points = (int)distance;
```

Το `(int)` μετατρέπει τη μεταβλητή `distance` σε `int`. Στην ουσία δεν αλλάζει ο τύπος της μεταβλητής `distance`, η οποία παραμένει `double`. Αλλάζει απλά ο τύπος της τιμής της, ώστε να μπορεί στη συνέχεια να αποθηκευτεί σε μία αντίστοιχου τύπου μεταβλητή, όπως εδώ η `points`. Άρα, η `points` είναι η «ακέραια έκδοση» της τιμής της `distance`.

ΠΡΟΣΟΧΗ: Η `double` τιμή της μεταβλητής `distance` δεν αλλάζει. Η `(int)` δημιουργεί μια ακέραια τιμή βάσει της `double` τιμής. Άρα δεν στρογγυλοποιείται η `double` τιμή στην αντίστοιχη `int`, αλλά απλά χάνεται το δεκαδικό της μέρος. Έτσι,

- αν `distance` = 12.29, `(int)distance` → 12
- αν `distance` = 12.99, `(int)distance` → 12 πάλι (απλά χάνεται το δεκαδικό μέρος και μένει το ακέραιο μέρος).

Γενικά, η σύνταξη μετατροπής τύπου είναι:

```
(τύπος)μεταβλητή;
ή
(τύπος)έκφραση;
```

Άρα, στο παράδειγμα του προηγούμενου μέρους που η διαίρεση των ακεραίων 3/4 έδωσε αποτέλεσμα 0 επειδή και το αποτέλεσμα είναι ακέραιος, εάν δε θέλαμε να χάσουμε την ακρίβεια της πράξης, θα έπρεπε να μετατρέψουμε τουλάχιστον τον έναν από τους δύο `int` σε `double`:

`(double)3 / 4` → `0.75` ή `3 / (double)4` → `0.75`

ή απλά να γράψουμε τον ένα αριθμό σε μορφή πραγματικού:

3.0 / 4 → 0.75 ή 3 / 4.0 → 0.75

Μετατροπή τύπου `char` σε `int` και το αντίστροφο:

```
char symbol;
symbol = '7';
System.out.println((int)symbol);  →  55
```

Τυπώνει 55 γιατί ο *χαρακτήρας* 7 της μεταβλητής `symbol` είναι ένας χαρακτήρας Unicode και το 55 είναι η θέση του στη λίστα των χαρακτήρων αυτών. Δηλαδή η μετατροπή σε `int` γίνεται μέσω της αντιστοιχίας του Unicode. Το `'7'` εδώ δεν είναι ένας αριθμός, είναι απλά ένας χαρακτήρας, ο no. 55 του συνόλου χαρακτήρων Unicode. Κάθε χαρακτήρας λοιπόν αντιστοιχεί σε έναν ακέραιο: τη θέση του στον πίνακα των χαρακτήρων Unicode. Χρειάζεται ιδιαίτερη προσοχή στη διαχείριση χαρακτήρων και στις πράξεις μεταξύ χαρακτήρων σε μια εντολή `System.out.println`.

Η εντολή `System.out.println('a');` θα τυπώσει: `a`
όμως η εντολή `System.out.println('a' + 'b');` θα τυπώσει `195`. Αυτό συμβαίνει γιατί με τον τελεστή + στην ουσία οι χαρακτήρες μετατρέπονται στους αντίστοιχους ακέραιους (97 και 98 αντίστοιχα) και έτσι πραγματοποιείται η πρόσθεση. [Εάν κάποιος, με αντίστοιχο τρόπο, ήθελε να εμφανίσει τους χαρακτήρες a και b στη σειρά (ab), θα έπρεπε να χρησιμοποιήσει συμβολοσειρές αντί για χαρακτήρες: `System.out.println("a" + "b");`. Αυτό εκτυπώνει: `ab`].

Η εντολή `System.out.println('a' + 1);` εκτυπώνει `98`. Εάν κάποιος θέλει να εκτυπώσει τον επόμενο χαρακτήρα του `'a'` και όχι το άθροισμα 97+1, τότε θα πρέπει να *μετατρέψει* το άθροισμα σε `char`: `System.out.println(`**`(char)`**`('a' + 1));`. Αυτό εκτυπώνει: `b`.

Άρα, το `(int)'χαρακτήρας'` αντιστοιχεί στον ακέραιο της θέσης του χαρακτήρα στο σύνολο Unicode, ενώ το `(char)ακέραιος` αντιστοιχεί στον χαρακτήρα που βρίσκεται στη συγκεκριμένη τιμή θέσης στο σύνολο Unicode. Σε περίπτωση που το `(char)` εφαρμοστεί σε πραγματικό αριθμό, λειτουργεί με το ακέραιο μέρος του.

Κανόνες προτεραιότητας πράξεων

Τρόπος για τον έλεγχο της προτεραιότητας: παρενθέσεις. Οι παρενθέσεις έχουν προτεραιότητα στις πράξεις, ξεκινώντας από τις πιο εσωτερικές. Όταν δεν ελέγχεται η προτεραιότητα από τον προγραμματιστή, η Java αποφασίζει βάσει συγκεκριμένων κανόνων, δηλαδή με βάση την ακόλουθη σειρά προτεραιότητας:

i) μοναδιαίοι τελεστές: - *(το πρόσημο)*, ++, --, ! *(η άρνηση - περισσότερα σε επόμενο μάθημα)*
ii) δυαδικοί τελεστές *, /, %
iii) δυαδικοί τελεστές +, -

(*μοναδιαίοι* τελεστές ονομάζονται αυτοί που εφαρμόζονται σε μία μόνο μεταβλητή, ενώ *δυαδικοί* αυτοί που εφαρμόζονται σε δύο μεταβλητές)

π.χ., το $\frac{a-b}{c+d}$ γράφεται: `(a - b) / (c + d)`

Αν το γράφαμε χωρίς παρενθέσεις, δηλαδή: `a - b / c + d` , τότε θα αντιπροσώπευε το $a-\frac{b}{c}+d$ (η διαίρεση b/c θα γινόταν πρώτη).

Άλλο παράδειγμα: $\frac{x}{y+5z}$ → `x / (y + 5 * z)` ή `x / (y + (5 * z))`

επειδή όμως ο πολλαπλασιασμός προηγείται της πρόσθεσης, η επιπλέον παρένθεση στο `(5*z)` δεν είναι απαραίτητη (χωρίς φυσικά να είναι λάθος εάν υπάρχει).

Μαθηματικές συναρτήσεις

Math.sin(x) → ημ(x), Math.cos(x) → συν(x), Math.exp(x) → e^x, Math.pow(x,y) → x^y

Math.sqrt(x) → √x, Math.abs(x) → |x|, Math.log(x) → $\log_e x$, Math.log10(x) → $\log_{10} x$, Math.PI → π

`Math.sin(x)` → ημ(x), `Math.cos(x)` → συν(x), `Math.exp(x)` → e^x,
`Math.pow(x,y)` → x^y, `Math.sqrt(x)` → √x, `Math.abs(x)` → |x|,
`Math.log(x)` → $\log_e x$, `Math.log10(x)` → $\log_{10} x$, `Math.PI` → π

Άλλο ένα παράδειγμα προγράμματος Java

Μας ζητείται να γράψουμε πρόγραμμα Java που να μετατρέπει ένα ποσό x λεπτών (cents) του ευρώ (όπου το x είναι ακέραιο ποσό μεταξύ 1 και 99) σε όσο το δυνατόν λιγότερα κέρματα των 50c, 20c, 10c, 5c, 2c και 1c (c = cents = λεπτά).

→ Τι μεταβλητές χρειαζόμαστε;

- έχουμε ένα αρχικό ποσό (1-99 cents), άρα θέλουμε μια `int` μεταβλητή: `originalAmount`
- θα μετράμε το πλήθος του κάθε κέρματος, άρα θέλουμε μια `int` μεταβλητή για κάθε κέρμα: `coin50`, `coin20`, `coin10`, `coin5`, `coin2` και `coin1`.

Αυτές προς το παρόν.

→ Πώς θα υπολογίσουμε τα κέρματα **(αλγόριθμος)**;

- Πόσα 50c υπάρχουν στο αρχικό ποσό;
- Τι ποσό απομένει αφού αφαιρέσουμε τα κέρματα των 50c;
  - Άρα χρειαζόμαστε κι άλλη μεταβλητή! Για να κρατάμε το ποσό που απομένει κάθε φορά. → μεταβλητή `amount`
- κτλ, για 20c, 10c, 5c, 2c, 1c.

Ο κώδικας Java:

```
import java.util.*;  // "βιβλιοθήκη" για user input
public class ChangeMaker
{
   public static void main(String [] args)
   {
      int originalAmount, amount, coin50, coin20, coin10, coin5, coin2, coin1;
      System.out.println("Δώσε το ποσό των λεπτών (από 1 ως 99)");
      Scanner input = new Scanner(System.in);
      originalAmount = input.nextInt();

      amount = originalAmount;
      coin50 = amount / 50;  // δουλεύει γιατί η amount είναι int
      amount = amount % 50;  // το υπόλοιπο ποσό
      coin20 = amount / 20;
      amount = amount % 20;
      coin10 = amount / 10;
      amount = amount % 10;
      coin5 = amount / 5;
      amount = amount % 5;
      coin2 = amount / 2;
      amount = amount % 2;
      coin1 = amount;        // το τελικό ποσό που απομένει

      System.out.println(originalAmount + " λεπτά σε κέρματα είναι:");
      System.out.println(coin50 + " 50λεπτα " + coin20 + " 20λεπτα " + coin10 +
       " 10λεπτα " + coin5 + " 5λεπτα " + coin2 + " 2λεπτα και " coin1 +
       " λεπτά.");

   }   // end of main

}   // end of class
```

Σύγκριση μεταβλητών

Η σύγκριση δύο μεταβλητών στη Java γίνεται με τους ακόλουθους τελεστές (δεξιά στήλη):

Μαθηματικά	*Java*
=	==
<	<
≤	<=
>	>
≥	>=
≠	!=

- Το αποτέλεσμα μιας σύγκρισης είναι τύπου `boolean` (`true` ή `false`)

π.χ.,

```
int a=5, b=5;
boolean c, d;
c = (a == b);  //  → c=true
d = (a > b);   //  → d=false
```

Λογικές σχέσεις

- Σύζευξη (AND): `&&`
- Διάζευξη (OR): `||`
- Άρνηση (NOT): `!`

- Λογικές σχέσεις γίνονται μεταξύ `boolean` μεταβλητών ή εκφράσεων που έχουν `boolean` αποτέλεσμα (τιμή).
- Το αποτέλεσμά τους είναι και αυτό `boolean` (`true` ή `false`).

Πίνακας αληθείας:

p	**q**	**p&&q**	**p||q**
True	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

- Ο υπολογισμός των λογικών σχέσεων σταματάει τη στιγμή που προσδιορίζεται η τιμή τους (υπάρχει τρόπος να *μη* γίνεται αυτό, με τα σύμβολα `&` και `|`). Δηλαδή, στη λογική σχέση: A&&B, αν το A είναι `false`, τότε δεν ελέγχεται καν το B, και το αποτέλεσμα του A&&B γίνεται `false`. Αντίστοιχα, στο A || B, αν το A είναι `true`, το αποτέλεσμα γίνεται `true` χωρίς να ελεγχθεί το B.

Άρα, η ακόλουθη έκφραση δε δημιουργεί πρόβλημα:

```
(3 == 7) && (2 == (3 / 0) );
```
→ βγάζει `false`

(γιατί 3 == 7 είναι `false` και δε μπαίνει ποτέ στο 3/0 του δεύτερου μέρους που κανονικά είναι σφάλμα (διαίρεση με το μηδέν)).

Αντίθετα, αν ήταν:

```
(7 == 7) && (2 == (3 / 0) );
```

θα έβγαζε *σφάλμα* (run-time error: division by zero).

Άρα, είναι επιθυμητό κατά τη δημιουργία λογικών σχέσεων, η απλούστερη έκφραση να γράφεται πρώτη, όπου αυτό είναι δυνατό.

Οι αντίστοιχοι τελεστές `&` και `|` ελέγχουν και τις δύο εκφράσεις, ανεξάρτητα από το αποτέλεσμα της πρώτης έκφρασης. Άρα, η έκφραση που παραπάνω έβγαζε `false`, με χρήση του `&` θα είναι εσφαλμένη, αφού θα ελέγξει και τη δεύτερη, λανθασμένη έκφραση:

`(3 == 7) & (2 == (3 / 0) );` → `run-time error: division by zero`

Συχνά λάθη:

- Το x δεν ισούται με 2 ή 3:

Λάθος: `x!=2 || x!=3` // → Είναι πάντα `true`!

Σωστό: `x!=2 && x!=3` // ή: `!(x==2 || x==3)`

Ισχύουν οι ιδιότητες: → Το: `!(p || q)` ταυτίζεται με το: `!p && !q`
→ Το: `!(p && q)` ταυτίζεται με το: `!p || !q`

- Το: $0 \le x < 1$ γράφεται: `0 <= x && x < 1`

(δεν συγκρίνουμε πάνω από δύο μεταβλητές μαζί)

Παράδειγμα:

- Για δεδομένο έτος (`year`) να δοθεί τιμή στη `boolean` μεταβλητή `isLeapYear` ανάλογα με το αν το έτος είναι δίσεκτο (`true`) ή όχι (`false`):

Λύση:

Πότε ένα έτος είναι δίσεκτο;
Δίσεκτα είναι τα έτη που διαιρούνται με το 4, *εκτός* από τα έτη των αιώνων (π.χ., 1700, 1800, 1900, κτλ.). Κατ' εξαίρεση, *είναι* δίσεκτα τα έτη των αιώνων που διαιρούνται με το 400 (π.χ., 1600, 2000, 2400, κτλ.).

Άρα, ένα έτος είναι δίσεκτο:

α) αν διαιρείται με το 4 αλλά όχι με το 100
ή
β) αν διαιρείται με το 400

Οπότε:

```
int year;
boolean isLeapYear;
.
.
isLeapYear = ((year%4==0) && (year%100!=0)) || (year%400==0);

// ή καλύτερα, πρώτα η απλούστερη έκφραση (year%400==0)
// και μετά η σύζευξη:
// isLeapYear = (year%400==0) || ((year%4==0) && (year%100!=0));
```

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 3

ΕΛΕΓΧΟΣ ΡΟΗΣ ΠΡΟΓΡΑΜΜΑΤΟΣ

I. Ελεγκτές συνθηκών ή περιπτώσεων:
- i) `if/else`
- ii) `switch`

II. Επαναληπτικές διαδικασίες:
- i) `for`
- ii) `while` (με έλεγχο συνθήκης)
- iii) `do/while` (με έλεγχο συνθήκης)

- **if/else:**

Με επιλογές μιας μόνο εντολής:

```
if (<λογική_έκφραση>)
   <εντολή_1>;  // if true
else
   <εντολή_2>;  // if false
```

Με επιλογές πολλαπλών εντολών:

```
if (<λογική_έκφραση>)
{
   <εντολή_1>;
   <εντολή_2>;
      ...
}
else
{
   <εντολή_Α>;
   <εντολή_Β>;
      ...
}
```

- `λογική_έκφραση` → `boolean`, δηλαδή `true` ή `false`
- Το `else` είναι προαιρετικό, άρα ισχύουν και τα:

```
if (<λογική_έκφραση>)
   <εντολή>;
```

ή

```
if (<λογική_έκφραση>)
{
   <εντολές>;
      ...
}
```

Π.χ.

```
int a=5, b=7, c;
if (a>=b)
   c = a+b;
else
   c = a-b;
System.out.println(c);   // → -2
```

Άλλα παραδείγματα:

- Μέγιστος μεταξύ των x και y:

```
if (x>y)                                    max = x;
   max = x;                                 if (y>x)
else                              ή           max = y;
   max = y;
```

- Απόλυτη τιμή του x:

```
if (x>=0)
   abs = x;                                 abs = x;
else                              ή         if (x<0)
   abs = -x;                                   abs = -x;
```

- Αν η `test` είναι μια `boolean` μεταβλητή:

```
if (test)
   <εντολή1>;
<εντολή2>;
<εντολή3>;
```

Σε αυτό το παράδειγμα δεν υπάρχει `else`. Επομένως μόνο η `<εντολή1>` «ανήκει» στο `if` και εκτελείται μόνο εάν η `test` είναι `true`. Η `<εντολή2>` και η `<εντολή3>` εκτελούνται ούτως ή άλλως, αφού βρίσκονται *μετά* το `if`.

Άρα: - Αν η `test` είναι `true`: `<εντολή1>, <εντολή2>, <εντολή3>`
- Αν η `test` είναι `false`: `<εντολή2>, <εντολή3>`

Όμως:

```
if (test)
   <εντολή1>;
else
   <εντολή2>;
<εντολή3>;
```

Σε αυτό το παράδειγμα η `<εντολή2>` ανήκει στην "`else`" περίπτωση του `if`, άρα εκτελείται *μόνο* εάν η `test` είναι `false`. Η `<εντολή3>` εκτελείται ούτως ή άλλως, αφού βρίσκεται μετά το `if`.

Άρα: - Αν η `test` είναι `true`: `<εντολή1>, <εντολή3>`
- Αν η `test` είναι `false`: `<εντολή2>, <εντολή3>`

- **Πρώτη διευκρίνιση:** Η στοίχιση των εντολών μπορεί πολλές να είναι παραπλανητική. Η Java δεν λαμβάνει υπ' όψιν της τη στοίχιση που κάνει ο προγραμματιστής στον κώδικά του και απλά διαχωρίζει τις εντολές της (π.χ. πού τελειώνει ένα `if` ή ένα `else`) σύμφωνα με τους κανόνες της. Π.χ., στο δεύτερο παράδειγμα παραπάνω, δεν θα άλλαζε τίποτα εάν κατά λάθος η `<εντολή3>` είχε γραφεί ακριβώς κάτω από την `<εντολή2>` ως εξής:

```
if (test)
   <εντολή1>;
else
   <εντολή2>;
   <εντολή3>;
```

Η στοίχιση αυτή είναι παραπλανητική, γιατί δεν αλλάζει τίποτα σε σχέση με πριν: στο `else` «ανήκει» μόνο η `<εντολή2>`, γιατί δεν υπάρχουν `{ }` άρα στο `else` ανήκει μόνο μία εντολή. Άρα η `<εντολή3>` παρόλο που φαίνεται να «ανήκει» στο `else`, είναι στην ουσία εκτός `if-else`, όπως ακριβώς πριν, και εκτελείται ούτως ή άλλως, ανεξαρτήτως της τιμής της `test`.

Αντίστοιχα, στο παρακάτω παράδειγμα:

```
if (test)
   <εντολή1>;
   <εντολή2>;
else
   <εντολή3>;
```

θα προκύψει σφάλμα (*'else' without 'if'*), αφού στο `if` ανήκει μόνο η `<εντολή1>`, ενώ η `<εντολή2>`, παρόλη την παραπλανητική στοίχισή της, είναι "αυτόνομη" και θα εκτελεστεί μετά το `if`, ενώ στη συνέχεια υπάρχει ένα `else` το οποίο δεν "ανήκει" σε κάποιο `if`, αφού το υπάρχον `if` έχει ολοκληρωθεί νωρίτερα.

- **Δεύτερη διευκρίνιση:** Προφανώς η «λογική έκφραση» ή η «συνθήκη» που ελέγχεται σε ένα `if` μπορεί να είναι και απλά μια `boolean` μεταβλητή, αφού και αυτή, όπως το αποτέλεσμα μιας λογικής έκφρασης, παίρνει τιμές `true` ή `false`. Άρα, στο προηγούμενο παράδειγμα, για τη `boolean` μεταβλητή `test`, αντί να γράψουμε:

```
if (test == true)
```

είναι το ίδιο να γράψουμε:

```
if (test)
```

αφού και στις δύο περιπτώσεις το αποτέλεσμα είναι `true` αν η `test` έχει την τιμή `true` και `false` αν η `test` έχει την τιμή `false`.

Αντίστοιχα, για τον έλεγχο:

```
if (test == false)
```

γράφουμε απλά:

```
if (!test)
```

Ένθετο if (nested)

a) Πολλαπλές περιπτώσεις

```
if (συνθήκη1)
{
   ...
}
else if (συνθήκη2)
{
   ...
}
else if (συνθήκη3)
{
   ...
}
else
{
   ...
}
```

b) if μέσα σε if

```
if (συνθήκη1)
{
   ...
   if (συνθήκη2)
   {
      ...
   }
   else
   {
      ...
   }
}
else
{
   ...
}
```

Παράδειγμα με if/else if και user input:

```
import java.util.*;
public class Grades
{
   public static void main(String [] args)
   {
      int score;   // η βαθμολογία στην κλίμακα 0-100
      char grade;  // ο βαθμός σαν A, B, C, D ή F

      Scanner input = new Scanner(System.in);
      System.out.println("Δώσε τη βαθμολογία σου (0-100):");
      score = input.nextInt();

      if (score > 90)
         grade = 'A';
      else if (score > 80)
         grade = 'B';
      else if (score > 70)
         grade = 'C';
      else if (score > 60)
         grade = 'D';
      else
         grade = 'F';

      System.out.println("Με βαθμολογία " + score +
                         " στα 100, ο βαθμός σου είναι: " + grade);

   }  // end of main
}  // end of class
```

● switch:

```
switch (varName)
{
  case value1:
     εντολές;
     break;
  case value2:
     εντολές;
     break;
  case value3:
     εντολές;
     break;
  default:
     εντολές;
}
```

→ Η μεταβλητή `VarName` είναι είτε ακέραια, είτε τύπου `boolean`, είτε τύπου `char`.

Το `switch` «στέλνει» τη ροή του κώδικα στο ισχύον case. Αν δεν υπάρχει κάποιο `break` στο τέλος ενός `case`, τότε η ροή του κώδικα συνεχίζει στο επόμενο `case`! Δηλαδή, κάνει έλεγχο των `case` μέχρι να βρει αυτό που ισχύει και από εκεί και πέρα δεν ελέγχει εάν τα υπόλοιπα ισχύουν ή όχι, αλλά απλά εκτελεί τον κώδικα μέχρι να βρει κάποιο `break` ώστε να βγει από το `switch`.

Παράδειγμα με switch:

```
char grade;
  .
  .  // απόδοση τιμής στη μεταβλητή grade
switch (grade)
{
  case 'A':
     System.out.println("Πολύ καλά!");
  case 'B':
     System.out.println("Μπράβο!");
     break;
  case 'C':
      System.out.println("Έτσι κι έτσι...");
      break;
  .
  .
}
```

Αν το grade είναι 'A' θα εκτυπώσει:

```
Πολύ καλά!
Μπράβο!
```

(γιατί δεν υπάρχει `break` στην 1η περίπτωση)

Αν το grade είναι 'B' θα εκτυπώσει:

```
Μπράβο!
```

Αν το grade είναι 'C' θα εκτυπώσει:

```
Έτσι κι έτσι...
```

Παράδειγμα με switch:

Να γραφεί πρόγραμμα που να ζητάει από το χρήστη τον αριθμό του μήνα (1-12) και να εκτυπώνει στην οθόνη το πλήθος των ημερών του μήνα αυτού, υποθέτοντας ότι το έτος δεν είναι δίσεκτο.

```
import java.util.*;
public class Ex1
{
   public static void main(String [] args)
   {
      Scanner input = new Scanner(System.in);
      System.out.println("Δώσε τον αριθμό του μήνα (1-12)");
      int month = input.nextInt();
      switch (month)
      {
         case 4:
         case 6:
         case 9:
         case 11:
         {
            System.out.println("Ο μήνας έχει 30 ημέρες.");
            break;
         }
         case 2:
         {
            System.out.println("Ο μήνας έχει 28 ημέρες.");
            break;
         }
         default:
            System.out.println("Ο μήνας έχει 31 ημέρες.");
      } // end switch
   } // end main
} // end class
```

- **for:** Επανάληψη μέρους του κώδικα για συγκεκριμένο αριθμό επαναλήψεων

Γενική σύνταξη:

```
for (αρχικοποίηση; συνθήκη; ανανέωση)
  εντολή;
```

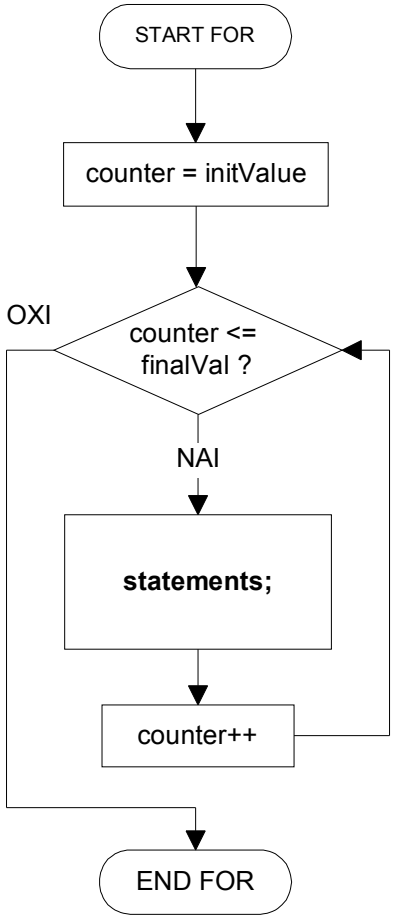
συνήθως:

```
for (int counter=initialValue; counter<=finalValue; counter++)
{
  εντολές;
}
```

Σειρά εκτέλεσης εντολών:

```
   i)    int counter = initialVal;
   ii)   έλεγχος counter<=finalValue ?
   iii)  εντολές;
   iv)   counter++
   v)    έλεγχος, κτλ.
```

Διάγραμμα Ροής του for-loop



→ Συνήθως ο μετρητής (counter) ορίζεται τοπικά, στο πρώτο μέρος της παρένθεσης του for-loop, δηλαδή είναι μια τοπική μεταβλητή. Παράδειγμα:

```
for (int i=1; i<=5; i++)
{
   System.out.println(i);
}
```

Τυπώνει:
```
1
2
3
4
5
```

Σε αυτό το παράδειγμα, εάν υπήρχε μια εντολή `System.out.println(i);` έξω από το for-loop **δεν** θα εκτύπωνε την τιμή 6 αλλά θα εμφάνιζε σφάλμα κατά το compilation του προγράμματος. Η μεταβλητή i που έχει δημιουργηθεί μέσα στο for είναι τοπική μεταβλητή του for και άρα "υπάρχει" μόνο μέσα στο μπλοκ του for (όπως αυτό ορίζεται από τα άγκιστρα { ... }).
Φυσικά ο μετρητής του for μπορεί να έχει ορισθεί πιο πάνω στο πρόγραμμα. Στην περίπτωση αυτή δεν θα έπρεπε να ξαναορισθεί μέσα στο for αλλά απλά να χρησιμοποιηθεί:

```
int i;
...
for (i=1; i<=5; i++)
  ...
```

και τότε θα είχε νόημα ένα `System.out.println(i)` εκτός του for.

→ Για να αυξάνεται ο μετρητής ενός for κατά 2 (αντί για 1 που είναι συνήθως το βήμα):

```
for (int i=0; i<10; i+=2)
```

και ομοίως για οποιοδήποτε άλλο βήμα.

Παράδειγμα: Υπολογισμός του n!

```
.
.
.
int n = nextInt();
int factorial = 1;

for (int i=1; i<=n; i++)
{
   factorial *= i;
}

System.out.println("Το " + n + " παραγοντικό είναι ίσο με " + factorial);
```

Nested for-loops (for μέσα σε for):

π.χ.

```
int k;
for (int i=0; i<3; i++)
{
    k=0;

    for (int j=0; j<5; j++)
    {
      k = k+j+i;
    }

    System.out.println(k);
}
```

Σειρά εκτέλεσης:
k=0;
5 φορές το "k=k+j+i;"
System.out.println(k);
2η φορά το "k=0;"
Άλλες 5 φορές το "k=k+j+i;"
κτλ. …

Άρα, θα εκτυπωθεί: **10**
15
20

Κάποια πιθανά σφάλματα στη δημιουργία ενός for:

- Ατέρμονες επαναλήψεις – Π.χ.:

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

- Καμία επανάληψη – Π.χ.:

```
for (int j=1; j<=0; j--)
{
      System.out.println(j);
}
```

Συμβουλή: Μπορούμε να χρησιμοποιήσουμε τον *μετρητή* του `for` σε πράξεις κτλ, αλλά καλό είναι να μην τον μεταβάλουμε (να μην αλλάζουμε την τιμή του) μέσα στο loop. Συνήθως το `for` χρησιμοποιείται για επανάληψη του κώδικα για συγκεκριμένο αριθμό φορών, ο οποίος καθορίζεται από τον τρόπο μεταβολής του μετρητή του `for`. Αν ο μετρητής μεταβάλλεται και μέσα στο loop, τότε η διαδικασία περιπλέκεται...

Παράδειγμα: Αριθμοί Fibonacci

Οι αριθμοί Fibonacci δίνονται από την ακολουθία: $f_1 = 1$, $f_2 = 1$, $f_n = f_{n-1} + f_{n-2}$, για $n>2$, ο καθένας δηλαδή είναι το άθροισμα των δύο προηγούμενών του: 1, 1, 2, 3, 5, 8, 13, 21, ...

Ένα πρόγραμμα Java που υπολογίζει συγκεκριμένο αριθμό αριθμών Fibonacci:

```
import java.util.*;
public class Fibonacci
{
   public static void main(String [] args)
   {
      Scanner input = new Scanner(System.in);
      System.out.println("Πόσοι αριθμοί Fibonacci?");
      int n = input.nextInt();

      int g=1, f=1;  // οι δύο πρώτοι αριθμοί Fibonacci

      System.out.println(g);
      System.out.println(f);

      for (int i=3; i<=n; i++)
      {
         f = f + g;
         g = f - g;  // Το g γίνεται ίσο με το προηγούμενο f
         System.out.println(f);
      }

   }
}
```

- **- while:**

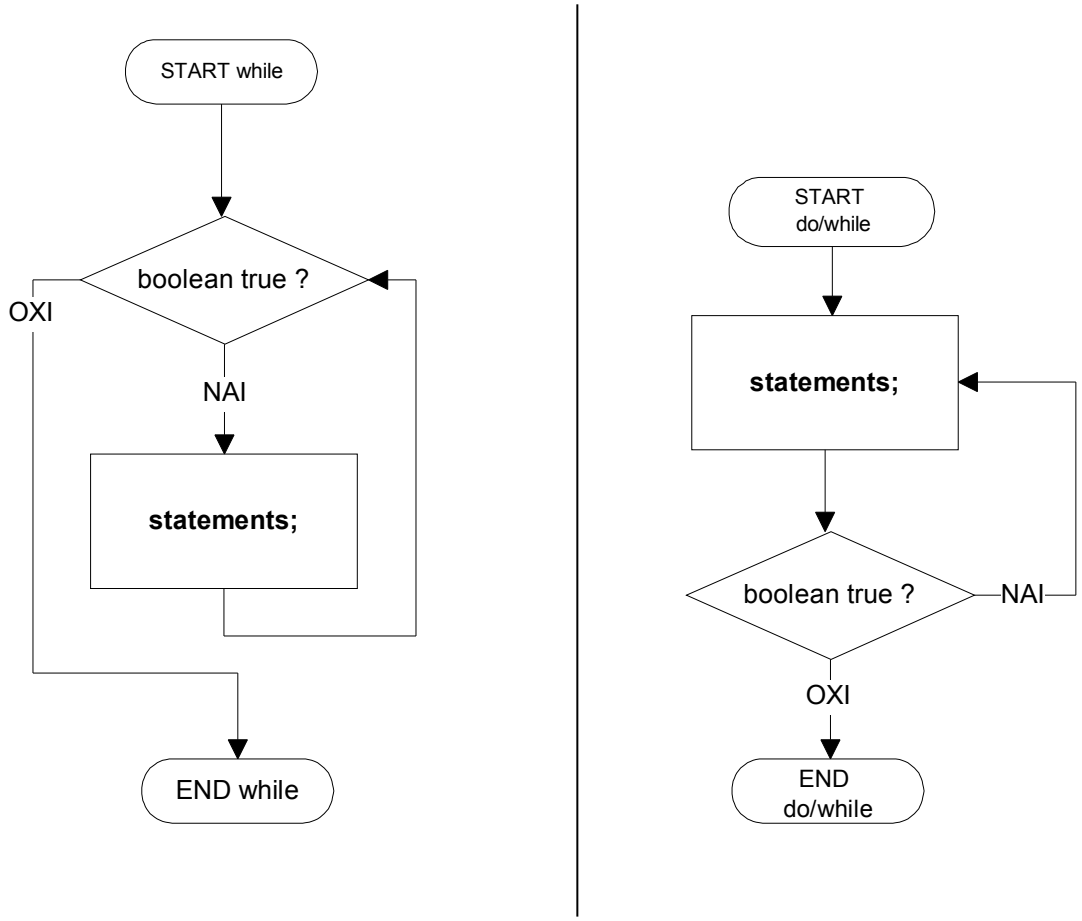
```
while(συνθήκη)
{
  εντολές;
}
```

- do/while:

```
do
{
  εντολές;
}while(συνθήκη);
```

Η συνθήκη είναι μία `boolean` έκφραση (άρα: `true` ή `false`).
Και το `while` και το `do/while` κάνουν επανάληψη των εντολών καθ' όσον η συνθήκη είναι αληθής. Η διαφορά μεταξύ τους είναι ότι το `do/while` θα εκτελέσει τις εντολές *τουλάχιστον μία φορά*, ανεξάρτητα με το αν είναι `true` η συνθήκη, αφού την ελέγχει στο τέλος του loop, ενώ η `while` δεν μπαίνει καθόλου στο loop αν αρχικά η συνθήκη είναι `false`.

<u>Διαγράμματα Ροής του `while` και του `do/while`</u>



Παραδείγματα while:

```
int n=0;
while (n<5)
{
 S.O.P.(n++);
}
```

Θα τυπώσουν:

0
1
2
3
4

```
int n=0;
while (n<5)
{
 S.O.P.(++n);
}
```

1
2
3
4
5

```
int n=0;
while (n<5)
{
 S.O.P.(n--);
}
```

0
-1
-2
-3
-4
...

(ατέρμονες επαναλήψεις)

```
int n=5;
while (n<5)
{
 S.O.P.(n--);
}
```

-

Άλλο παράδειγμα while:

- Άθροισμα ψηφίων ακεραίου:

Αλγόριθμος: i) εύρεση τελευταίου ψηφίου: n%10 (π.χ., 1975%10 → 5)
ii) αποκοπή τελευταίου ψηφίου: n/10 (π.χ., 1975/10 → 197)

```
...
int n = input.nextInt();
int dsum = 0;
while (n>0)
{
   dsum += n%10;
   n /= 10;
}
System.out.println(dsum);
```

Σχέση for και while:

```
for (αρχικοποίηση; συνθήκη; ανανέωση)
{
   εντολές;
}
```

```
αρχικοποίηση;
while(συνθήκη)
{
    εντολές;
    ανανέωση;
}
```

π.χ.

```
for (int i=1; i<=5; i++)
{
   System.out.println(i);
}
```

```
int i=1;
while(i<=5)
{
  System.out.println(i);
  i++;
}
```

For και while για εισαγωγή αριθμών από το πληκτρολόγιο

Π.χ., θέλουμε να εισάγουμε αριθμούς και να υπολογίσουμε το άθροισμά τους:

i) Για *συγκεκριμένο πλήθος* αριθμών (**for**):

```
double total = 0, value;
System.out.println("Πλήθος?");
int n = input.nextInt();
for (int i=1; i<=n; i++)
{
   System.out.println("Τιμή?");
   value = input.nextDouble();
   total += value;
}
System.out.println(total);
```

ii) Για *άγνωστο πλήθος* αριθμών (**while**):

```
double endOfData = 0;
double total = 0, value;
System.out.println("Τιμή?");
value = input.nextDouble();
while (value != endOfData)
{
   total += value;
   System.out.println("Τιμή? – Δώσε 0 για τερματισμό");
   value = input.nextDouble();
}
System.out.println(total);
```

Αντίστοιχα, με τη χρήση **do-while**:

```
double endOfData = 0;
double total = 0, value;
do
{
   System.out.println("Τιμή? – Δώσε 0 για τερματισμό");
   value = input.nextDouble();
   total += value;
} while (value != endOfData);
System.out.println(total);
```

Σημείωση 1: Η μεταβλητή `endOfData` στο παραπάνω παράδειγμα χρησιμοποιείται για τον τερματισμό της διαδικασίας εισόδου αριθμών από το πληκτρολόγιο (οι μεταβλητές που χρησιμοποιούνται για αυτόν ή κάποιον αντίστοιχο σκοπό, λέγονται “μεταβλητές flag”). Στο συγκεκριμένο παράδειγμα, η διαδικασία τερματίζεται όταν ο χρήστης δώσει την τιμή 0. Η τιμή της `endOfData` επιλέγεται από τον προγραμματιστή (π.χ., 0, -1, 999, κτλ.). Ο χρήστης θα πρέπει φυσικά να ενημερώνεται για την τιμή της (με ανάλογο μήνυμα) ώστε να γνωρίζει πώς να τερματίσει τη διαδικασία εισόδου δεδομένων.

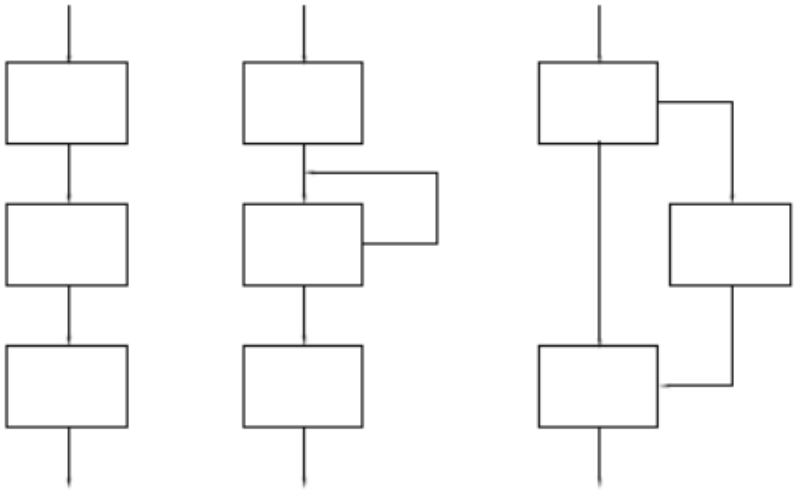
Σημείωση 2: Με τη χρήση του do-while αποφεύχθηκε η επανάληψη του κώδικα “ερώτησης-εισόδου από το πληκτρολόγιο”, η οποία έγινε αναγκαστικά στην περίπτωση του while για να δοθεί από το πληκτρολόγιο η πρώτη τιμή. Κάποιες φορές αυτό μπορεί να μην είναι δυνατό ή επιθυμητό.

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 4

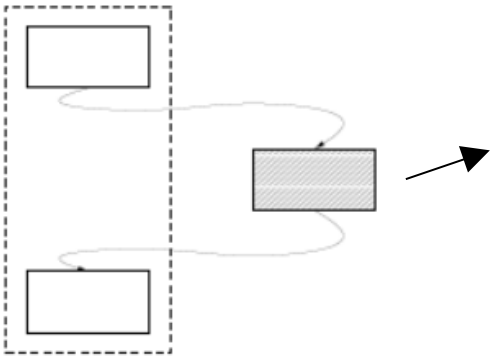
Αντικειμενοστραφής Προγραμματισμός (Μέθοδοι, Κλάσεις, Αντικείμενα)

Με τον όρο «ροή προγράμματος» αναφερόμαστε στη σειρά με την οποία εκτελούνται οι εντολές ενός προγράμματος. Μέχρι τώρα έχουμε αναφερθεί στις ακόλουθες μορφές ροής:

Σειριακή Επαναληπτική Υπό συνθήκη



Μια άλλη μορφή ροής ενός προγράμματος είναι αυτή που πραγματοποιείται μέσω της κλήσης μεθόδων:



μέθοδος (συνάρτηση) που μπορεί να βρίσκεται μέσα στην κλάση ή και μέσα σε κάποια άλλη κλάση.

Μέθοδοι

Μέθοδος ονομάζεται ένα σύνολο συγκεντρωμένων εντολών με χαρακτηριστικό όνομα. (Είναι το αντίστοιχο των «συναρτήσεων» άλλων γλωσσών προγραμματισμού)

Η φιλοσοφία των μεθόδων:

- Εντοπισμός κάποιου υπο-προβλήματος που χρειάζεται να επιλυθεί ως μέρος του προγράμματος
- Αλγοριθμική επίλυση του υπο-προβλήματος και εγγραφή του αντίστοιχου κώδικα (μόνο μία φορά)
- Ονομασία του κώδικα του υπο-προβλήματος → μέθοδος
- Χρήση της μεθόδου (κλήση χρησιμοποιώντας το όνομά της → εκτέλεση) κάθε φορά που παρουσιάζεται το ίδιο υπο-πρόβλημα.

Βασικά χαρακτηριστικά μεθόδων:

- Επαναχρησιμοποιήσιμος κώδικας σε άλλα προγράμματα
- Μη επανάληψη κώδικα στο ίδιο πρόγραμμα
- «Μαύρα κουτιά» → ενδιαφέρει το τι κάνουν, όχι το πώς το κάνουν
- Βοηθούν στη λογική σχεδίαση του προγράμματος
- Αποκρύπτουν προγραμματιστικές λεπτομέρειες

Είδη μεθόδων:

- Ορισμένες από τον προγραμματιστή
- Βιβλιοθήκες/Πακέτα (μέρος της Java), π.χ. println(), nextInt(), sqrt() κτλ.

Σχεδιασμός μεθόδων:

- Κάθε μέθοδος καλείται από κάποια άλλη μέθοδο
- Μπορεί να έχει δεδομένα εισόδου (τα παίρνει από τη μέθοδο που την καλεί)
- Μπορεί να έχει δεδομένα εξόδου (τα επιστρέφει στη μέθοδο που την καλεί)
- Έχει ένα όνομα (συνήθως μοναδικό, αλλά όχι πάντα → βλ. Ενότητα 5)
- Έχει ένα σύστημα ορισμού της επικοινωνίας της με το περιβάλλον

Τα δύο τελευταία δηλώνονται με τον ορισμό της μεθόδου στην πρώτη της γραμμή στον κώδικα.

Ορισμός μεθόδων:

```
[ορατότητα] <τύπος> <όνομα> ([παράμετροι])
     ↓          ↓
  public      void
  private     int
     .       double
     .         .
     .         .
```

Οι παράμετροι, εάν υπάρχουν, είναι π.χ. της μορφής:

```
... <όνομαΜεθόδου> (<τύπος> <παράμ1>, <τύπος> <παράμ2>, ...)
{
    ... x = παράμ1;
    ... y = 5 * παράμ2;
          ...
}
```

Τι είναι το κάθε στοιχείο του ορισμού:

i) **ορατότητα** (public/private): ορίζει το από ποιο μέρος του προγράμματος είναι ορατή η μέθοδος.
public: η μέθοδος μπορεί να κληθεί και από μεθόδους άλλων κλάσεων, όχι μόνο της κλάσης μέσα στην οποία βρίσκεται.
private: η μέθοδος μπορεί να κληθεί μόνο από μεθόδους της ίδιας της κλάσης στην οποία βρίσκεται. *[Περισσότερα για την ορατότητα, αργότερα...]*

ii) **τύπος**: Μία μέθοδος μπορεί να επιστρέφει *το πολύ* μία τιμή.
- void εάν η μέθοδος δεν επιστρέφει τίποτα
- πρωτογενής τύπος (int, double, boolean, char, κτλ.) ανάλογα με τον τύπο της μεταβλητής που επιστρέφει η μέθοδος
- String εάν η μέθοδος επιστρέφει κάποια συμβολοσειρά

iii) **παράμετροι:** μεταβλητές εισόδου, οι οποίες έχουν συγκεκριμένες τιμές στη μέθοδο από την οποία καλείται η συγκεκριμένη μέθοδος, και χρησιμοποιούνται με αυτές τις τιμές μέσα στην καλούμενη μέθοδο.
(Αν δεν υπάρχουν παράμετροι εισόδου, οι παρενθέσεις απλά μένουν κενές)

Η εντολή return:

- Χρησιμοποιείται στις μεθόδους συγκεκριμένου τύπου (δηλ. όχι void) για να επιστρέψει την τιμή της μεταβλητής εξόδου:

```
            return μεταβλητή;
      ή     return έκφραση;
```

π.χ.
```
public int method1(int x)
{
   int y = 2*x;      // θα μπορούσε να γραφεί κατευθείαν:
   return y;         // return 2*x;
}
```

- Χρησιμοποιείται στις `void` μεθόδους ως εντολή εξόδου από τη μέθοδο (και επιστροφής της ροής του κώδικα στην αρχική μέθοδο κλήσης).
  Η χρήση αυτή είναι σπάνια, συνήθως χρησιμοποιείται υπό συνθήκη και καλό είναι να αποφεύγεται (υπάρχουν καλύτεροι τρόποι που επιτυγχάνουν ανάλογο αποτέλεσμα)

Κλήση μεθόδων (μέσα από την ίδια κλάση):

- Αν η μέθοδος είναι `void`:

```
<όνομαΜεθόδου>([ορίσματα]);
```

όπου, ορίσματα είναι συγκεκριμένες τιμές για τις παραμέτρους εισόδου που τυχόν έχει η μέθοδος. Εάν οι παράμετροι εισόδου είναι περισσότερες από μία, τότε τα ορίσματα πρέπει να δοθούν με τη σειρά, με αντιστοίχιση θέσης (το 1ο όρισμα εκχωρείται στην 1η παράμετρο εισόδου, το 2ο όρισμα στη 2η παράμετρο εισόδου, κοκ).

- Αν η μέθοδος δεν είναι `void`:

```
<μεταβλητή> = <όνομαΜεθόδου>([ορίσματα]);
```

Ένα απλό παράδειγμα μιας κλάσης με δύο μεθόδους (κλήση μεθόδου μέσα στην ίδια κλάση):

(Η μέθοδος `square` ορίζεται ως `static` για να μπορεί να την καλέσει απ' ευθείας η μέθοδος `main` η οποία είναι και αυτή `static`. Η λέξη `static` καθώς και το τι ακριβώς συμβαίνει με τις `static` μεθόδους και μεταβλητές, θα αναφερθούν σε επόμενη ενότητα, μετά της εισαγωγή στον αντικειμενοστραφή τρόπο προγραμματισμού)

```
import java.util.*;
public class Example
{
   // Η μέθοδος square:
   public static double square(double x)
   {
      double s = x*x;
      return s;
   }

   // Η μέθοδος main:
   public static void main(String [] args)
   {
      Scanner input = new Scanner(System.in);

      System.out.println("Δώσε έναν αριθμό:");
      double n = input.nextDouble();

      double z = square(n);

      System.out.println(n + " στο τετράγωνο: " + z);
   }

}
```

Παράδειγμα εκτέλεσης του προγράμματος:

```
> Δώσε έναν αριθμό:
> 5
> 5 στο τετράγωνο: 25
>
```

Άλλο ένα παράδειγμα με κλήση μεθόδου στην ίδια κλάση:

```
import java.util.*;
public class Circle
{

   //μέθοδος που υπολογίζει εμβαδόν κύκλου:
   public static double circleArea(double r)
   {
      return Math.PI*r*r;
   }

   // μέθοδος που υπολογίζει εμβαδόν δακτυλίου:
   public static double ringArea(double rIn, double rOut)
   {
      double innerArea = circleArea(rIn);
      double outterArea = circleArea(rOut);
      return (outterArea – innerArea);
   }

   // Η μέθοδος main:

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

      double area;

      System.out.println("Ακτίνα 1?");
      double r1 = input.nextDouble();
      System.out.println("Ακτίνα 2?");
      double r2 = input.nextDouble();

      if (r1 < r2)
         area = ringArea(r1,r2);
      else
         area = ringArea(r2,r1);

      System.out.println("Το εμβαδόν του δακτυλίου είναι: " + area);
   }

}
```

ΚΛΑΣΗ: σύνθετος τύπος δεδομένων

Οι κλάσεις αποτελούν τη βασική δομή του αντικειμενοστραφούς προγραμματισμού. Ουσιαστικά μια κλάση είναι ο ορισμός ενός σύνθετου τύπου δεδομένων από τον προγραμματιστή. Οι μεταβλητές που υλοποιούν έναν τέτοιο σύνθετο τύπο δεδομένων (μια κλάση) ονομάζονται **αντικείμενα** (objects)[1]. Όπως δηλαδή οι απλές μεταβλητές είναι κάποιου συγκεκριμένου πρωτογενούς τύπου (π.χ., `int`, `double` κτλ), έτσι και τα αντικείμενα είναι σύνθετες μεταβλητές, τύπου κάποιας κλάσης.

Η μεγάλη διαφορά μεταξύ των πρωτογενών τύπων και των κλάσεων είναι ότι οι πρωτογενείς τύποι προορίζονται για τη δημιουργία απλών μεταβλητών που αποθηκεύουν ένα απλό δεδομένο (π.χ., έναν ακέραιο αριθμό, έναν χαρακτήρα κτλ), ενώ οι κλάσεις είναι πιο πολύπλοκες και προορίζονται για τη δημιουργία οντοτήτων (αντικειμένων) τα οποία "ομαδοποιούν" πολλά δεδομένα (χαρακτηριστικά) καθώς και συναρτήσεις (συμπεριφορές).

Όλα τα αντικείμενα που έχουν κοινά χαρακτηριστικά ανήκουν στην ουσία στην ίδια κλάση. Η κλάση είναι ένα "καλούπι" που ορίζει αυτά τα χαρακτηριστικά, και το κάθε αντικείμενο που δημιουργείται με βάση αυτή την κλάση έχει τη δυνατότητα να "αποθηκεύσει" συγκεκριμένες τιμές για καθένα από αυτά τα χαρακτηριστικά που ορίζει η κλάση του.

Έχει αναφερθεί σε προηγούμενη ενότητα ότι οι τύποι (μεταβλητών, μεθόδων, κτλ.) στη Java μπορούν γενικά (για διδακτικούς σκοπούς) να χωριστούν σε τρεις κατηγορίες:

Τύποι (types):

- Πρωτογενείς (primitive types) (`int`, `double`, `char`, κτλ.)
- String (συμβολοσειρές (αλλιώς: αλφαριθμητικά))
- **Κλάσεις** → σύνθετη δομή σχεδιασμένη από τον προγραμματιστή

Όπως μια μεταβλητή ορίζεται (δηλώνεται) ως:

```
<τύπος> <μεταβλητή>;
```

και π.χ. μια μεταβλητή `a` μπορεί να είναι ακέραια, δηλαδή να είναι μια μεταβλητή τύπου `int`:

```
int a;
```

έτσι και ένα αντικείμενο ορίζεται (δηλώνεται) ως:

```
<Κλάση> <αντικείμενο>;
```

και π.χ. ένα αντικείμενο `ferrari` μπορεί να είναι της κλάσης `Car` , δηλαδή να είναι τύπου `Car`:

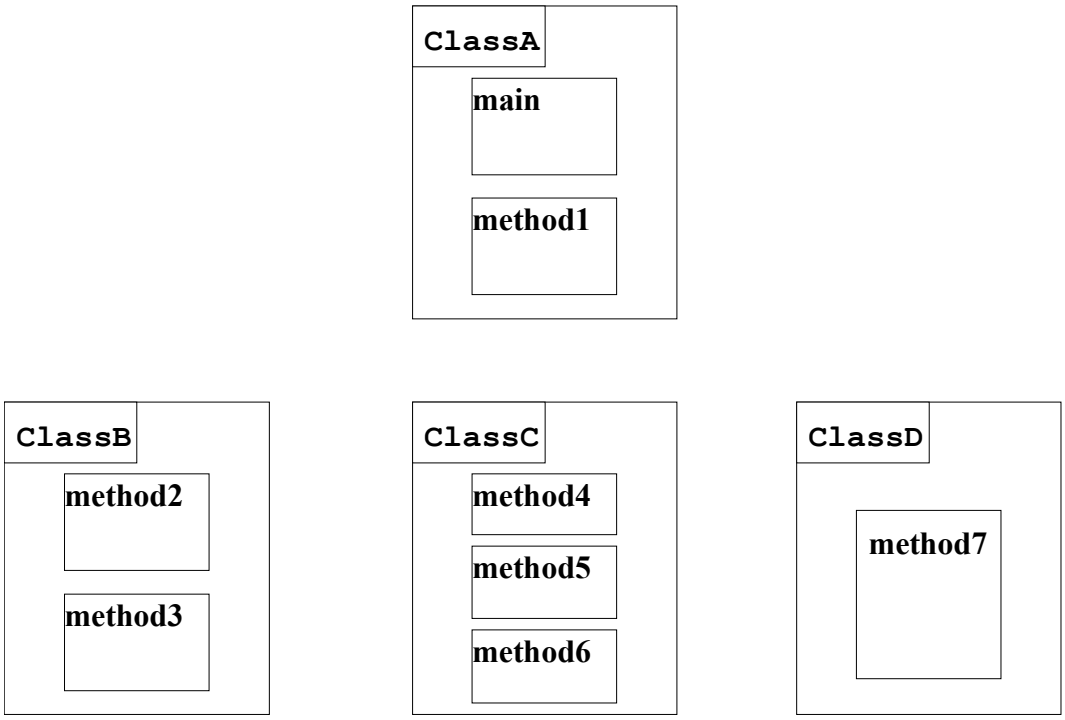
```
Car ferrari;
```

Προσοχή: Δεν αρκεί αυτή η εντολή για τη δημιουργία του αντικειμένου. Η εντολή αυτή απλά δηλώνει το αντικείμενο ως αντικείμενο του συγκεκριμένου τύπου (εδώ τύπου `Car`). *Περισσότερα για αυτό, αργότερα.*

1 Ένα αντικείμενο (object) λέγεται και στιγμιότυπο (instance) μιας κλάσης – εδώ θα χρησιμοποιείται ο όρος αντικείμενο.

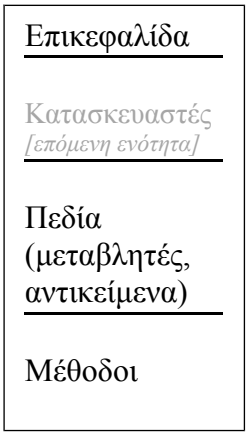
Δηλαδή, ενώ μια *μεταβλητή* είναι "κάποιου πρωτογενούς τύπου", ένα *αντικείμενο* είναι "τύπου κάποιας κλάσης". Ουσιαστικά, όπως προαναφέρθηκε, τα αντικείμενα είναι "σύνθετες μεταβλητές" και άρα οι κλάσεις είναι "σύνθετοι τύποι δεδομένων". *(Οι μεταβλητές τύπου String είναι στην ουσία αντικείμενα, αφού η String είναι μια κλάση. Περισσότερα για αυτό, στην Ενότητα 6).*

Ένα πρόγραμμα μπορεί να έχει πολλές κλάσεις και κάθε κλάση μπορεί να έχει πολλές μεθόδους. Η μέθοδος `main` είναι πάντα *μία* σε ένα πρόγραμμα. Η κλάση που την περιέχει λέγεται κλάση εφαρμογής. Οι υπόλοιπες κλάσεις λέγονται κλάσεις υποστήριξης. Δηλαδή, ένα πρόγραμμα μπορεί να έχει την ακόλουθη δομή:



Κλάση εφαρμογής: `ClassA`
Κλάσεις υποστήριξης: `ClassB, ClassC, ClassD`

ΔΟΜΗ ΚΛΑΣΗΣ:




```
public class <ΟνομαΚλάσης>
{
      .
      .
      .

}
```

Παράδειγμα Κλάσης (υποστήριξης):

Επικεφαλίδα: >
Πεδία: >
>
Μέθοδος: >
>
>

```
public class ToKelvin
{
    private double C2K = 273.15;  //μεταβλητή

    public double returnKelvin (double tempC) //μέθοδος
    {
        return (tempC + C2K);
    }
}
```

(Πλήρης) ορισμός μεταβλητής:

```
[ορατότητα] <τύπος> <όνομα> [= <τιμή>];
     ↓          ↓
  public      int
  private   double
     .          .
     .          .
```

```
π.χ.: public double var1;
      private int var2 = 5;
```

Οι μεταβλητές αυτές ορίζονται ***εκτός μεθόδων*** *(συνήθως στην αρχή της κλάσης)* και ονομάζονται ***μεταβλητές αντικειμένου***, αφού ανήκουν σε κάθε αντικείμενο της κλάσης, και προφανώς σε όλες τις μεθόδους της (δηλαδή μπορούν να χρησιμοποιηθούν από όλες τις μεθόδους της κλάσης). Οι μεταβλητές που ορίζονται *μέσα* σε μεθόδους ανήκουν μόνο στη μέθοδο στην οποία ορίζονται και ονομάζονται ***τοπικές μεταβλητές***. Έτσι, ορίζεται η έννοια της *εμβέλειας*:

Εμβέλεια μεταβλητής (scope): τα σημεία μιας κλάσης στα οποία υφίσταται κάποια μεταβλητή (δηλ., εάν είναι μεταβλητή αντικειμένου ή τοπική μεταβλητή (μιας μεθόδου ή ενός βρόχου κτλ.)).

Ορατότητα (μεταβλητής ή μεθόδου): ορίζει σε ποιες κλάσεις ενός προγράμματος μπορεί να χρησιμοποιηθεί μια μέθοδος ή μια μεταβλητή. Όταν π.χ. μια μεταβλητή έχει ορατότητα `private` μπορεί να χρησιμοποιηθεί μόνο από μεθόδους της κλάσης στην οποία ορίζεται, ενώ αν έχει ορατότητα `public` μπορεί να χρησιμοποιηθεί από όλες τις κλάσεις του προγράμματος. Υπάρχουν και άλλες μορφές ορατότητας (που έχουν να κάνουν με την κληρονομικότητα και τα πακέτα), οι οποίες θα αναφερθούν σε επόμενες ενότητες.

Άρα, μία κλάση στη γενική της μορφή έχει τη δομή:

```
public class <ΌνομαΚλάσης>
{
      μεταβλητές αντικειμένου;

      μέθοδοι()

}
```

Τύποι μεθόδων:

Στον *αντικειμενοστραφή προγραμματισμό*, διακρίνουμε τρεις βασικούς τύπους μεθόδων:

1) **Λειτουργικές μέθοδοι (Operation methods):**

```
public void opMethod()
{
    //δηλώσεις;
    //υπολογισμοί;
}
```

(είναι πάντα `void`, δεν επιστρέφουν τίποτα, απλά υπολογίζουν κάτι)

2) **Τροποποιητικές μέθοδοι (Modifier methods ή setter)**

```
private double var;

public void modMethod(double x)
{
   var = x;
}
```

(δέχονται πάντα κάποια παράμετρο εισόδου (π.χ. `x`), την τιμή της οποίας «περνάνε» σε κάποια `private` μεταβλητή αντικειμένου της κλάσης τους (π.χ. `var`))

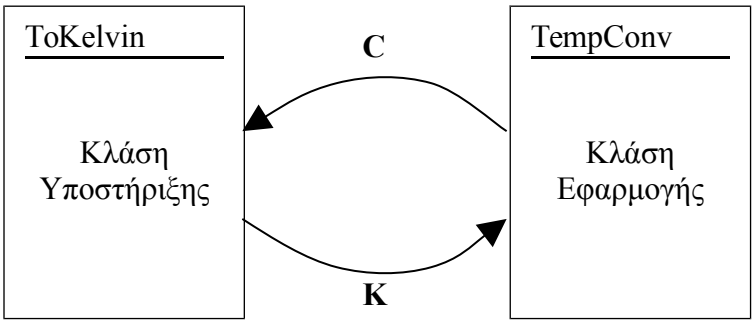
3) **Μέθοδοι πρόσβασης (Accessor methods ή getter)**

```
private int var2;

public int accessMethod()
{
   return var2;
}
```

(είναι πάντα `public`, επιστρέφουν (`return`) μια `private` μεταβλητή και είναι του ίδιου τύπου με τη μεταβλητή που επιστρέφουν – στο παραπάνω παράδειγμα `int`)

Συχνά υπάρχουν μέθοδοι που είναι συνδυασμοί των παραπάνω τύπων, κυρίως των τύπων (2) και (3).

Η κλάση `ToKelvin` του παραπάνω παραδείγματος είναι μια *κλάση υποστήριξης*. Για να ολοκληρωθεί το πρόγραμμα πρέπει να φτιάξουμε μια *κλάση εφαρμογής* (`TempConv`), έτσι ώστε η κλάση εφαρμογής να "στέλνει" βαθμούς Κελσίου στην κλάση υποστήριξης και η κλάση υποστήριξης με τη σειρά της να τους μετατρέπει σε βαθμούς Kelvin και να τους επιστρέφει στην κλάση εφαρμογής:



Για να γίνει η σύνδεση μεταξύ των δύο κλάσεων ώστε η μία να μπορεί να καλεί μεθόδους της άλλης, θα πρέπει η πρώτη να δημιουργήσει αντικείμενο της δεύτερης.

Δημιουργία αντικειμένου μιας κλάσης:

```
<Κλάση>  <αντικείμενο>  =  new  <Κλάση>( );
```

Ουσιαστικά η εντολή αυτή είναι δύο εντολές μαζί:

i) Δήλωση αντικειμένου (που αναφέρθηκε πιο πριν):

```
<Κλάση>  <αντικείμενο>;
```

ii) Δημιουργία αντικειμένου *(περισσότερα για αυτό το βήμα στην επόμενη ενότητα)*:

```
<αντικείμενο> = new <Κλάση>();
```

Στο συγκεκριμένο παράδειγμα, η κλάση `TempConv` που θα είναι η κλάση εφαρμογής (θα έχει τη μέθοδο `main`), θα χρειαστεί να καλέσει μεθόδους της κλάσης υποστήριξης (`ToKelvin`). Επομένως θα πρέπει πρώτα να δημιουργήσει αντικείμενο της κλάσης `ToKelvin`:

Η κλάση εφαρμογής:

```
public class TempConv
{
   public static void main (String [ ] args)
   {
        // δημιουργία (τοπικών) μεταβλητών:
      double C = 15.5;  // μια θερμοκρασία σε βαθμούς Κελσίου
      double K;

        // δημιουργία αντικειμένου της κλάσης ToKelvin:
      ToKelvin toK = new ToKelvin();

        // κλήση της μεθόδου returnKelvin με αποστολή  της θερμοκρασίας
        // σε βαθμούς Κελσίου και επιστροφή της σε βαθμούς Kelvin:
      K = toK.returnKelvin(C);

        // εκτύπωση στην οθόνη της θερμοκρασίας, σε Kelvin:
      System.out.println(K);

   }  // end main

}  // end class
```

Σε αυτό το παράδειγμα, βλέπουμε τον τρόπο κλήσης μιας μεθόδου από μια μέθοδο κάποιας άλλης κλάσης. Αναλυτικά:

Κλήση μεθόδου του αντικειμένου μιας κλάσης:

Προϋπόθεση είναι να έχει δημιουργηθεί το αντικείμενο της κλάσης στην οποία βρίσκεται η μέθοδος

```
<αντικείμενο>.<μέθοδος>( );
```

Αν η μέθοδος δεν είναι `void`, δηλαδή αν επιστρέφει κάτι, μπορεί να κληθεί (και συνήθως καλείται) ως εξής:

```
<μεταβλητή>  =  <αντικείμενο>.<μέθοδος>( );
```

Παράδειγμα από την παραπάνω κλάση `(TempConv)`:

```
K = toK.returnKelvin(C);
```

Παραδείγματα:

Κλάση υποστήριξης:

```
public class ClassA
{
     // μεταβλητές:
   private int a = 5;
   private int b;
   private double c;

     // τροποποιητική μέθοδος:
   public void setB(int x)
   {
      b = x;
   }

     // λειτουργική μέθοδος:
   public void calcC()
   {
      c = 0.5*(a+b);
   }

     // μέθοδος πρόσβασης:
   public double getC()
   {
      return c;
   }
}
```

Κλάση εφαρμογής:

```
public class ClassB
{
   public static void main (String [ ] args)
   {
      // δημιουργία 4 αντικειμένων της ClassA:
      ClassA obj1 = new ClassA();
      ClassA obj2 = new ClassA();
      ClassA obj3 = new ClassA();
      ClassA obj4 = new ClassA();

      // κλήση διαφόρων μεθόδων για τα αντικείμενα:
      obj1.setB(10);
      obj1.calcC();
      obj2.setB(20);
      obj2.calcC();
      obj3.setB(15);
      obj4.calcC();

      // εκτύπωση στην οθόνη:
      System.out.println(obj1.getC());  // θα τυπώσει: 7.5
      System.out.println(obj2.getC());  // θα τυπώσει: 12.5
      System.out.println(obj3.getC());  // θα τυπώσει: 0
      System.out.println(obj4.getC());  // θα τυπώσει: 2.5

   }  // end method

}  // end class
```

Το `obj1.getC()` είναι 7.5 γιατί έχουμε καλέσει πριν τη μέθοδο `setB` για το συγκεκριμένο αντικείμενο (`obj1`) στέλνοντας στο `b` την τιμή 10, οπότε το `c` υπολογίζεται σαν 0.5*(5+10). Ομοίως για το `obj2.getC()`, μόνο που τώρα η τιμή του `b` είναι 20, οπότε το `c` γίνεται 12.5. Στην περίπτωση του `obj3.getC()`, η τιμή είναι 0 γιατί *δεν* έχουμε καλέσει τη μέθοδο `calcC` *για το αντικείμενο αυτό* (`obj3`), οπότε δεν έχει υπολογιστεί κάποια τιμή για το `c` αυτού του αντικειμένου. Αντίστοιχα, στην περίπτωση του αντικειμένου `obj4`, δεν στάλθηκε κάποια τιμή στη μεταβλητή `b` του αντικειμένου αυτού (μέσω της `setB(...)`), οπότε το `c` που υπολογίζεται είναι το 0.5*(5+0) = 2.5.

Έχοντας δηλώσει **`private`** τις μεταβλητές της κλάσης υποστήριξης (ClassA) και κυρίως τη μεταβλητή `c`, **δεν** επιτρέπεται στην `ClassB` να αλλάξει την τιμή τους όπως θέλει, άρα π.χ. η `c` αλλάζει *μόνο* όπως ο προγραμματιστής έχει επιλέξει στην `ClassA` (μέσω της `calcC()`). Αυτός θεωρείται **ασφαλής τρόπος προγραμματισμού** και αποτελεί μία από τις πιο βασικές αρχές του αντικειμενοστραφούς προγραμματισμού, την **ενθυλάκωση** (encapsulation).

Ένα άλλο χαρακτηριστικό παράδειγμα που δείχνει ότι οι μέθοδοι και οι μεταβλητές μιας κλάσης ανήκουν στην ουσία στα αντικείμενά της:

Αν έχουμε μια κλάση υποστήριξης:

```
public class Dog
{
  private String color;

  public void setColor(String a)
  {
    color = a;
  }

  public String accessColor()
  {
    return color;
  }
}
```

τότε από κάποια άλλη κλάση υποστήριξης ή από την κλάση εφαρμογής, μπορούμε να φτιάξουμε αντικείμενα της κλάσης `Dog`, να στείλουμε τα χρώματα των αντικειμένων αυτών, ή από κάπου αλλού να «διαβάσουμε» τα χρώματά τους:

```
// δημιουργία αντικειμένων της κλάσης Dog
Dog fidel = new Dog();
Dog ivan = new Dog();

// αποστολή του χρώματος του κάθε αντικειμένου
fidel.setColor("καφέ");
ivan.setColor("μαύρο");
  .
  .
  .
// Όταν κάπου αλλού θέλουμε να ανακτήσουμε τα χρώματα
// των αντικειμένων που έχουμε φτιάξει:

System.out.println("Το χρώμα του Φιντέλ είναι " + fidel.accessColor());
System.out.println("Το χρώμα του Ιβάν είναι " + ivan.accessColor());
```

και αυτό που θα εκτυπωθεί στην οθόνη είναι το εξής:

```
Το χρώμα του Φιντέλ είναι καφέ
Το χρώμα του Ιβάν είναι μαύρο
```

Δηλαδή, η μεταβλητή `color` της κλάσης `Dog`, στην ουσία δεν έχει κάποια συγκεκριμένη τιμή. Συγκεκριμενοποιείται μόνο όταν φτιάξουμε κάποιο αντικείμενο της κλάσης `Dog`, και έχει τόσα «αντίγραφα» όσα είναι τα αντικείμενα. Αφού συγκεκριμενοποιηθεί από τη δημιουργία αντικειμένου, το κάθε «αντίγραφό» της παίρνει συγκεκριμένη τιμή όταν καλέσουμε τη μέθοδο `setColor`.

Ένα άλλο παράδειγμα *(η κλάση Complex για μιγαδικούς αριθμούς)*

Στη Java δεν υπάρχει ειδικός τύπος για μεταβλητές *μιγαδικών αριθμών*. Αν θέλουμε να μπορούμε να διαχειριστούμε μιγαδικούς αριθμούς, θα πρέπει να φτιάξουμε ένα δικό μας "σύνθετο τύπο" που να περιέχει τα στοιχεία και τις πράξεις μιγαδικών αριθμών, δηλαδή μία **Κλάση**. Τα βασικά στοιχεία μιας τέτοιας κλάσης θα μπορούσαν να είναι τα εξής:

```
public class Complex
{
  private double real, imag; // το πραγματικό και το φανταστικό μέρος

  public void setValues(double x, double y)
  {
    real = x;
    imag = y;
  }

  public double getReal()
  {
    return real;
  }

  public double getImaginary()
  {
    return imag;
  }

  public String printComplex()
  {
    if (imag>0)
      return (real + " + " + imag + "i");
    else if (imag<0)
      return (real + " - " + (-imag) + "i");
    else
     return (real + ""); //προσθέτουμε κενό String για να μετατραπεί όλο σε String
  }
}
```

Μέχρι στιγμής, η κλάση για τους μιγαδικούς αριθμούς περιέχει τα εξής:

- μία μέθοδο για ανάθεση τιμών στο πραγματικό και στο φανταστικό μέρος ενός μιγαδικού (`setValues`)
- μία μέθοδο που δίνει πρόσβαση στην τιμή του πραγματικού μέρους ενός μιγαδικού (`getReal`)
- μία μέθοδο που δίνει πρόσβαση στην τιμή του φανταστικού μέρους ενός μιγαδικού (`getImaginary`)
- μία μέθοδο που επιστρέφει έναν μιγαδικό αριθμό στη μορφή $a + b\,i$ (`printComplex`)

Για την υλοποίηση άλλων ιδιοτήτων των μιγαδικών αριθμών, όπως π.χ. πράξεων μεταξύ μιγαδικών κτλ., απαιτούνται στοιχεία της Java που δεν έχουν διδαχθεί ακόμη. Επομένως, το παράδειγμα της κλάσης `Complex` θα συνεχιστεί σε επόμενο μάθημα. Με τα στοιχεία που έχει όμως μέχρι στιγμής η κλάση, μπορούμε να την χρησιμοποιήσουμε για να δίνουμε τιμές σε μιγαδικούς αριθμούς και να τους εμφανίζουμε στην οθόνη. Αυτό υλοποιείται π.χ., με την ακόλουθη *κλάση εφαρμογής*:

```
import java.util.*;
public class TestComplex
{
    public static void main(String [] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.print("Δώσε το πραγματικό μέρος: ");
        double a = input.nextDouble();
        System.out.print("Δώσε το φανταστικό μέρος: ");
        double b = input.nextDouble();
        Complex c = new Complex();
        c.setValues(a, b);
        System.out.print("Ο μιγαδικός αριθμός είναι: ");
        System.out.println(c.printComplex());
    }
}
```

Παραδείγματα εκτέλεσης του προγράμματος:

```
Δώσε το πραγματικό μέρος: 2.4
Δώσε το φανταστικό μέρος: 4.6
Ο μιγαδικός αριθμός είναι: 2.4 + 4.6i

Δώσε το πραγματικό μέρος: 5
Δώσε το φανταστικό μέρος: -8.3
Ο μιγαδικός αριθμός είναι: 5.0 – 8.3i
```

Συνοψίζοντας, οι 4 γενικές περιπτώσεις μεθόδων (ως προς την είσοδο/έξοδο) και οι τρόποι κλήσης τους από μέθοδο της ίδιας κλάσης ή από μέθοδο άλλης κλάσης είναι οι εξής:

Μέθοδος στην κλάση MyClass	**Κλήση από τη MyClass**	**Κλήση από άλλη κλάση**

`public void method1()`
`{`
`  ...`
`}`
	`method1();`	`obj.method1();`

`public void method2(int x)`
`{`
`  ...`
`}`
	`int a = 5;`
	`method2(a);`
		`int b = 3;`
		`obj.method2(b);`

`public double method3()`
`{`
`  ...`
`}`
	`double x = method3();`	`double y = obj.method3();`

`public boolean method4(char c)`
`{`
`  ...`
`}`
	`char c = 'd';`
	`boolean b = method4(c);`
		`char d = 'a';`
		`boolean x = obj.method4(d);`

(Τα ονόματα και οι τιμές μεταβλητών καθώς και οι τύποι παραμέτρων εισόδου ή μεταβλητών επιστροφής των μεθόδων, είναι αυθαίρετα. ***Το `obj` είναι αντικείμενο της `MyClass`****. Επίσης, οι κλήσεις των μεθόδων της* `MyClass` *γίνονται προφανώς μέσα από άλλες μεθόδους.*

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 5

Κατασκευαστές (Constructors)

Ειδικός τύπος μεθόδων, οι οποίες:

- είναι `public` και έχουν το ίδιο όνομα με αυτό της κλάσης
- χρησιμοποιούνται για να αρχικοποιήσουν κάποιες μεταβλητές των αντικειμένων που δημιουργούν, κατά τη δημιουργία των αντικειμένων
- καλούνται αυτόματα όταν δημιουργείται ένα αντικείμενο κάποιας κλάσης
- δεν επιστρέφουν κάποια τιμή (αλλά παρόλα αυτά *δεν* δηλώνονται ως `void`)

Σε μία κλάση με το όνομα `ClassName` μπορούμε να έχουμε:

```
public class ClassName
{
   private int a;
   private String b;

     // default κατασκευαστής:
   public ClassName()
   {

   }

     // ένας άλλος κατασκευαστής:
   public ClassName(int x, String s)
   {
      a = x;
      b = s;
   }

   // άλλες δηλώσεις, μέθοδοι, κτλ.

}
```

- Κάθε κλάση περιέχει (αυτόματα από τη Java) τον default (προεπιλεγμένο) κατασκευαστή, παρόλο που δεν φαίνεται στον κώδικά της.

- Ο default κατασκευαστής παύει να δίνεται αυτόματα από τη στιγμή που δημιουργηθεί έστω και ένας κατασκευαστής από τον προγραμματιστή. Ο λόγος που γίνεται αυτό είναι ότι όταν ο προγραμματιστής δημιουργεί δικούς του κατασκευαστές που αρχικοποιούν κάποιες μεταβλητές, πιθανότατα να μην θέλει να δίνει το δικαίωμα κατασκευής αντικειμένων χωρίς την αρχικοποίηση αυτών των μεταβλητών. Εάν θέλουμε να μπορούμε να φτιάχνουμε αντικείμενα της κλάσης με τον απλό τρόπο (χωρίς παραμέτρους εισόδου για αυτόματη αρχικοποίηση μεταβλητών) θα πρέπει να προσθέσουμε στον κώδικα της κλάσης τον default κατασκευαστή, έστω και κενό.

Παράδειγμα κατασκευαστών:

```
public class Circle
{
   private double x, y;  // συντεταγμένες κέντρου
   private double r;     // ακτίνα

      // μέθοδος για υπολογισμό εμβαδού κύκλου
   public double area()
   {
      return Math.PI*Math.pow(r,2);
   }

      // μέθοδος για υπολογισμό περιφέρειας κύκλου
   public double circumf()
   {
      return 2*Math.PI*r;
   }
}
```

Οι μεταβλητές `x`, `y` και `r` είναι `private`. Ένας τρόπος για να τους δώσουμε τιμές θα ήταν η ύπαρξη αντίστοιχων τροποποιητικών (modifier) μεθόδων στην κλάση `Circle`. Επειδή όμως **κάθε** αντικείμενο («κύκλος») που θα δημιουργούμε αναγκαστικά θα πρέπει να έχει τουλάχιστον κάποια δεδομένη ακτίνα, μπορούμε να αυτοματοποιήσουμε την απόδοση τιμών σε (κάποιες από) αυτές τις μεταβλητές, με τη δημιουργία κάποιων κατασκευαστών:

```
public Circle(double a, double b, double c)
{
   x = a;
   y = b;
   r = c;
}

public Circle(double r)
{
   x = 0;
   y = 0;
   this.r = r;     // το this. αναφέρεται στη μεταβλητή της κλάσης
                   // ενώ πλέον το r είναι η τοπική μετ/τή της μεθόδου
}

public Circle()
{
   x = 0;
   y = 0;
   r = 1;
}
```

[Οι κατασκευαστές αυτοί τοποθετούνται φυσικά μέσα στην κλάση Circle, μετά τις δηλώσεις των μεταβλητών και συνήθως πριν τις μεθόδους της κλάσης]

Ο πρώτος κατασκευαστής αρχικοποιεί όλες τις μεταβλητές, ο δεύτερος αρχικοποιεί μόνο την ακτίνα (θέτοντας συγχρόνως τη θέση του κύκλου στο σημείο (0,0)), ενώ ο τρίτος είναι παραλλαγή του default constructor και δημιουργεί «μοναδιαίους κύκλους» (αντικείμενα με r = 1).

Έτσι, μπορούμε να έχουμε τις εξής δηλώσεις για τη δημιουργία αντικειμένων της κλάσης `Circle` (π.χ. μέσα στην κλάση εφαρμογής):

```
Circle c1 = new Circle(2.5, 3.5, 6);
```

(δημιουργεί «κύκλο» με κέντρο στο (2.5,3.5) και ακτίνα 6)

```
Circle c2 = new Circle(4.3);
```

(δημιουργεί «κύκλο» με κέντρο στο (0,0) και ακτίνα 4.3)

```
Circle c3 = new Circle();
```

(δημιουργεί μοναδιαίο «κύκλο» στο (0,0))

~~`Circle c4 = new Circle(1.3, 4.5);`~~ → ΛΑΘΟΣ! Δεν έχουμε φτιάξει κατασκευαστή που να δέχεται **δύο** παραμέτρους.

Όταν σε μια κλάση υπάρχουν περισσότεροι του ενός κατασκευαστές, τότε παρατηρείται το φαινόμενο του **overloading κατασκευαστών** (υπερφόρτωση). Αυτό συμβαίνει όταν στην ίδια κλάση βρίσκονται μέθοδοι με το ίδιο όνομα (στην προκειμένη περίπτωση κατασκευαστές, που αναγκαστικά έχουν όλοι το ίδιο όνομα) με διαφορετικό πλήθος ή/και είδος παραμέτρων εισόδου. Το φαινόμενο της υπερφόρτωσης θα αναλυθεί αργότερα σε αυτή την ενότητα, στην *υπερφόρτωση μεθόδων*. Ένα παράδειγμα ολοκληρωμένης κλάσης εφαρμογής λοιπόν:

```
public class RunCircle
{
    public static void main(String [] args)
    {
        Circle c1 = new Circle(2.5, 3.5, 6);
        Circle c2 = new Circle(4.3);
        Circle c3 = new Circle();

        System.out.println("Το εμβαδόν του 1ου κύκλου είναι: " + c1.area());
        System.out.println("Η περιφέρεια του 2ου κύκλου είναι: " + c2.circumf());
        System.out.println("Το εμβαδόν του 3ου κύκλου είναι: " + c3.area());
    }
}
```

Εκτύπωση προγράμματος:

```
Το εμβαδόν του 1ου κύκλου είναι: 113.09733552923255
Η περιφέρεια του 2ου κύκλου είναι: 27.01769682087222
Το εμβαδόν του 3ου κύκλου είναι: 3.141592653589793
```

Η λέξη `this`:

Η δεσμευμένη λέξη `this` μέσα σε μια μέθοδο, χρησιμοποιείται για να προσδιορίσει κάτι που ανήκει στο αντικείμενο με το οποίο καλείται η μέθοδος. Άρα, όταν χρησιμοποιείται με τη μορφή `this.μεταβλητή`, προσδιορίζει τη μεταβλητή αντικειμένου που έχει ορισθεί ως πεδίο της κλάσης. Εάν δεν υπάρχει κάποια τοπική μεταβλητή μέσα στη μέθοδο (ή κάποια παράμετρος εισόδου) με το ίδιο όνομα με μια μεταβλητή αντικειμένου, τότε δεν υπάρχει λόγος χρήσης τού `this`, αφού αρκεί η αναφορά στο όνομα της μεταβλητής μέσα στην κλάση. Όταν όμως υπάρχει τοπική μεταβλητή με το ίδιο όνομα με μια μεταβλητή αντικειμένου της κλάσης, τότε το όνομα της μεταβλητής μέσα στη μέθοδο αναφέρεται *στην τοπική μεταβλητή* (δηλαδή η τοπική μεταβλητή έχει προτεραιότητα στη

χρήση του κοινού ονόματος), και για να γίνει αναφορά στην αντίστοιχη μεταβλητή αντικειμένου, απαιτείται η χρήση της λέξης `this` ως: `this.μεταβλητή`.

Η λέξη `static`:

Όλα τα στοιχεία μιας κλάσης ανήκουν στα αντικείμενά της. Δηλαδή, μια μεταβλητή `x` μιας κλάσης `MyClass`, έχει τόσα αντίγραφα (τόσες υποστάσεις) όσα (-ες) και τα αντικείμενα της κλάσης `MyClass`. Το ίδιο ισχύει και για τις μεθόδους. Δηλαδή, κάθε αντικείμενο έχει τις δικές του μεταβλητές και μεθόδους, όπως αυτές ορίζονται στην κλάση του (στο «καλούπι» του).

Για τον περιορισμό αυτής της «ποικιλομορφίας» του αντικειμενοστραφούς προγραμματισμού, υπάρχει η λέξη `static`. Εξαίρεση λοιπόν στα προηγούμενα αποτελούν οι μεταβλητές και μέθοδοι που δηλώνονται ως **`static`**. Αυτές είναι κοινές για όλα τα αντικείμενα μιας κλάσης και δεν χρειάζεται η δημιουργία αντικειμένου για τη χρήση τους:

Οι `static` μέθοδοι καλούνται με τη μορφή:

```
<Κλάση>.<μέθοδος>();
```

αντί της κανονικής μορφής `<αντικείμενο>.<μέθοδος>();`

→ Μέθοδοι που έχουν δηλωθεί `static` έχουν πρόσβαση μόνο σε `static` μεταβλητές και `static` μεθόδους *της κλάσης τους*.

Άρα, η `main` (που είναι `static`) διαχειρίζεται μόνο `static` μεταβλητές κλάσης και αντικείμενα (και φυσικά τις δικές της τοπικές μεταβλητές, δηλαδή αυτές που έχουν δηλωθεί μέσα στην ίδια τη `main`) και μπορεί να καλέσει άμεσα μόνο `static` μεθόδους της κλάσης εφαρμογής. Υπάρχει τρόπος να ξεπεραστεί αυτό το "πρόβλημα". Ας δούμε το ακόλουθο παράδειγμα, που αποτελεί τροποποίηση μιας άσκησης της Ενότητας 3 που κάνει χρήση του `switch`:

- Να γραφεί μέθοδος που να δέχεται τον αριθμό του μήνα (1-12) και να επιστρέφει το πλήθος των ημερών του μήνα αυτού, υποθέτοντας ότι το έτος δεν είναι δίσεκτο.

```
public class Ex2
{
   public int daysInMonth(int month)
   {
      switch (month)
      {
         case 4:
         case 6:
         case 9:
         case 11: return 30;
         case 2: return 28;
         default: return 31;
      }
   }  // end method
}   // end class
```

Μέχρι το σημείο αυτό έχουμε ολοκληρώσει τη μέθοδο που ζητάει η άσκηση. Αν θέλαμε να ολοκληρώσουμε το πρόγραμμα ώστε να μπορεί να εκτελεστεί, θα πρέπει να προσθέσουμε τη μέθοδο `main` στην κλάση μας (`Ex2`), ως εξής (πριν ή μετά τη μέθοδο `daysInMonth`, δεν έχει σημασία):

```
   public static void main(String [] args)
   {
      Scanner input = new Scanner(System.in);
      Ex2 obj = new Ex2();
      System.out.println("Δώσε τον αριθμό του μήνα (1-12)");
      int m = input.nextInt();
         // κλήση της μεθόδου daysInMonth για υπολογισμό ημερών
      int d = obj.daysInMonth(m);
      System.out.println("Ο μήνας έχει " + d + " ημέρες.");
   }
```

και φυσικά δεν πρέπει να ξεχάσουμε να κάνουμε `import java.util.*;` στην κλάση μας (`Ex2`), αφού πλέον χρησιμοποιούμε τη `Scanner` για είσοδο από το πληκτρολόγιο.

ΣΗΜΑΝΤΙΚΟ: Σε αυτή τη μέθοδο `main` βλέπουμε ότι δημιουργούμε αντικείμενο της ίδιας της κλάσης μέσα στην οποία βρισκόμαστε (`Ex2`), και μέσω αυτού καλούμε τη μέθοδο `daysInMonth`. Ο λόγος είναι ότι ενώ η `main` είναι `static` μέθοδος, η `daysInMonth` δεν είναι `static`. Οι `static` μέθοδοι έχουν άμεση πρόσβαση μόνο σε `static` μεταβλητές και μεθόδους. Για να μπορέσει η `main` να καλέσει μια μη-`static` μέθοδο, πρέπει να την καλέσει μέσω κάποιου συγκεκριμένου αντικειμένου. Όταν καλεί μεθόδους άλλων κλάσεων (υποστήριξης), αυτό είναι αυτονόητο. Όταν όμως καλεί μη-`static` μεθόδους της κλάσης στην οποία βρίσκεται, δεν είναι προφανές ότι πρέπει να δημιουργήσει αντικείμενο της ίδιας της κλάσης της, γι αυτό και επισημαίνεται εδώ.

Μέθοδοι που δέχονται ή/και επιστρέφουν αντικείμενα

Τα αντικείμενα είναι σύνθετοι τύποι μεταβλητών, οπότε οι μέθοδοι μπορούν να τα διαχειριστούν (δηλαδή, να τα δεχθούν ή να τα επιστρέψουν) όπως και τις μεταβλητές. Όσον αφορά μάλιστα την επιστροφή μιας μεθόδου, με τη χρήση επιστροφής αντικειμένου (αντί για επιστροφή μιας απλής μεταβλητής) μια μέθοδος μπορεί ουσιαστικά να επιστρέφει περισσότερες από μία μεταβλητές, επιστρέφοντας στην ουσία όλες τις "μεταβλητές κλάσης" που μπορεί να περιέχει το συγκεκριμένο αντικείμενο που επιστρέφει. Ας δούμε το ακόλουθο παράδειγμα:

```
public class ClassA
{
  private int x, y;

   // κατασκευαστές
  public ClassA(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public ClassA()
  { }

   // μέθοδος απόδοσης τιμών στις μεταβλητές κλάσης
  public void setValues(int a, int b)
  {
    x = a;
    y = b;
  }
```

```
    // μέθοδοι πρόσβασης για τις δύο private μεταβλητές της κλάσης
  public int getX()
  {
    return x;
  }

  public int getY()
  {
    return y;
  }

    // μέθοδος που αυξάνει τις τιμές των x και y κατά 1
  public ClassA increaseXY(ClassA a)
  {
    ClassA ca = new ClassA(a.getX()+1, a.getY()+1);
    return ca;
  }
}
```

Στο παράδειγμα αυτό, η μέθοδος `increaseXY` της κλάσης `ClassA` δέχεται μια "μεταβλητή" τύπου `ClassA`, δηλαδή ένα αντικείμενο της κλάσης `ClassA` και επιστρέφει αντικείμενο της κλάσης `ClassA`. Αυτό που κάνει είναι να δημιουργεί και να επιστρέφει ένα νέο αντικείμενο της `ClassA` με τιμές των `x` και `y` αυτού του νέου αντικειμένου τις τιμές που έχουν οι μεταβλητές για το αντικείμενο `a` που δέχεται η μέθοδος αυξημένες κατά 1. Άρα, ουσιαστικά η συγκεκριμένη μέθοδος επιστρέφει *δύο* μεταβλητές και όχι μία, αφού κάθε αντικείμενο της κλάσης `ClassA` περιέχει δύο μεταβλητές.

Η δημιουργία του νέου αντικειμένου `ca` μέσα στη μέθοδο `increaseXY` αντί για:

```
ClassA ca = new ClassA(a.getX()+1, a.getY()+1);
```

θα μπορούσε να γραφεί και ως εξής:

```
ClassA ca = new ClassA(a.x+1, a.y+1);
```

Παρόλο που οι μεταβλητές `x` και `y` είναι `private`, αφού βρισκόμαστε μέσα στην κλάση `ClassA`, μπορούμε να έχουμε πρόσβαση στις τιμές τους απλά γράφοντας `<αντικείμενο>.<μεταβλητή>`, κάτι το οποίο δεν το χρησιμοποιούμε σχεδόν ποτέ, αφού πάντα ορίζουμε τις μεταβλητές κλάσης `private`, οπότε ο τρόπος αυτός πρόσβασης σε μια μεταβλητή ενός αντικειμένου *από κάποια άλλη κλάση* δεν είναι δυνατός.

ΠΡΟΣΟΧΗ: Εάν η δημιουργία αντικειμένου μέσα στην `increaseXY` γινόταν ως εξής:

```
ClassA ca = new ClassA(x+1, y+1);
```

ή

```
ClassA ca = new ClassA(this.x+1, this.y+1);  // είναι το ίδιο
```

η μέθοδος δεν θα έκανε αυτό το οποίο θα θέλαμε να κάνει. Αυτό που θα έκανε θα ήταν να αυξάνει την τιμή των `x` και `y` του αντικειμένου της `ClassA` *με το οποίο θα την καλούσαμε* και όχι του συγκεκριμένου αντικειμένου που δέχεται η μέθοδος (`a`).

Σε μία άλλη κλάση, η οποία χρησιμοποιεί την `ClassA`, θα μπορούσαν να υπάρχουν τα εξής:

```
public class Program
{
   public static void main (String[] args)
   {
      ClassA obj1 = new ClassA(5,10);
      ClassA obj2 = new ClassA(15,20);
      ClassA obj3 = obj1.increaseXY(obj2);

      System.out.println("Τιμές των x και y για το obj1:");
      System.out.println(obj1.getX());
      System.out.println(obj1.getY());

      System.out.println("Τιμές των x και y για το obj3:");
      System.out.println(obj3.getX());
      System.out.println(obj3.getY());
   }
}
```

Το πρόγραμμα θα εκτύπωνε τα εξής:

```
Τιμές των x και y για το obj1:
5
10
Τιμές των x και y για το obj3:
16
21
```

Οι τιμές 16 και 21 προκύπτουν από την αύξηση κατά ένα των τιμών των μεταβλητών του αντικειμένου `obj2`, που επιτυγχάνει η κλήση της μεθόδου `increaseXY` στην οποία αποστέλλεται (περνιέται) το αντικείμενο `obj2`. Προφανώς στο συγκεκριμένο παράδειγμα δεν έχει σημασία το αντικείμενο το οποίο χρησιμοποιείται για να γίνει η κλήση της μεθόδου (δηλαδή το `obj1`). Αν όμως η μέθοδος `increaseXY` περιείχε την εντολή που αναφέρθηκε προηγουμένως:

```
   ClassA ca = new ClassA(x+1, y+1);
```

τότε θα συνέβαινε το ακριβώς αντίθετο, δηλαδή η αύξηση κατά ένα θα γινόταν στις μεταβλητές του αντικειμένου με το οποίο θα καλούνταν η μέθοδος (δηλαδή το `obj1`) και όχι στου αντικειμένου που αποστέλλεται ως όρισμα στη μέθοδο (`obj2`). Άρα το πρόγραμμα θα εμφάνιζε το "λανθασμένο" αποτέλεσμα:

```
Τιμές των x και y για το obj1:
5
10
Τιμές των x και y για το obj3:
6
11
```

Άρα, τελικά η συγκεκριμένη μέθοδος ίσως να ήταν προτιμότερο να είχε δηλωθεί ως `static` ώστε να καλείται γενικά και όχι για κάποιο συγκεκριμένο αντικείμενο.

Παράδειγμα με την κλάση Complex

Στην προηγούμενη ενότητα δημιουργήθηκε ένας "σύνθετος τύπος" μεταβλητών (δηλαδή μια κλάση) για τους μιγαδικούς αριθμούς (`Complex`). Το περιεχόμενο της κλάσης αυτής ήταν το εξής:

```
public class Complex
{
  private double real, imag; // το πραγματικό και το φανταστικό μέρος
  public void setValues(double x, double y)
  {
    real = x;
    imag = y;
  }
  public double getReal()
  {
    return real;
  }
  public double getImag()
  {
    return imag;
  }
  public String printComplex()
  {
    if (imag>0)
      return (real + " + " + imag + "i");
    else if (imag<0)
      return (real + " - " + (-imag) + "i");
    else
     return (real + ""); //προσθέτουμε κενό String για να μετατραπεί όλο σε String
  }
}
```

→ Κάποιες βελτιώσεις και επεκτάσεις που μπορούν να γίνουν σε αυτή την κλάση:

1. Η διαδικασία απόδοσης τιμής στο πραγματικό και φανταστικό μέρος ενός μιγαδικού (μεταβλητές `real` και `imag` αντίστοιχα) μπορεί να αυτοματοποιηθεί μέσω ενός κατάλληλου κατασκευαστή (οπότε η ύπαρξη της μεθόδου `setValues` δεν θα είναι πλέον απαραίτητη):

```
  // Constructor
public Complex(double x, double y)
{
   real = x;
   imag = y;
}
```

i) Σε ποια περίπτωση θα παρέμενε απαραίτητη η ύπαρξη της μεθόδου `setValues`; → Στην περίπτωση που εκτός από τον παραπάνω κατασκευαστή, προσθέταμε και τον default constructor, οπότε κάποιος θα μπορούσε να φτιάξει αντικείμενο με τον κλασικό τρόπο χωρίς να στείλει κατευθείαν τιμές στις μεταβλητές (αν δεν προστεθεί ο default κατασκευαστής, η δυνατότητα αυτή χάνεται, από τη στιγμή που δημιουργεί κάποιος άλλος κατασκευαστής).

ii) Σε ποια άλλη περίπτωση θα εξακολουθούσε να είναι απαραίτητη η ύπαρξη της μεθόδου `setValues`; → Στην περίπτωση που θα θέλαμε να δώσουμε τη δυνατότητα δημιουργίας μεθόδων που δέχονται αντικείμενο τύπου `Complex` και μεταβάλουν τις τιμές των μεταβλητών του αντικειμένου αυτού. Μια τέτοια μέθοδος θα δεχόταν ένα ήδη

δημιουργημένο αντικείμενο, οπότε θα χρειαζόταν την ύπαρξη μιας τέτοιας μεθόδου για να μεταβάλει τα δεδομένα του αντικειμένου αυτού. (Το συγκεκριμένο παράδειγμα θα χρησιμοποιηθεί στην επόμενη ενότητα, που αναφέρεται στο "pass-by-value").

2. Μπορούν να προστεθούν μέθοδοι που επιτελούν πράξεις μεταξύ μιγαδικών αριθμών. Π.χ., για την πρόσθεση μιγαδικών θα μπορούσαν να ορισθούν οι ακόλουθες πιθανές εκδοχές μιας μεθόδου `add`:

```
  public Complex add(Complex a)
  {
     return new Complex(real+a.getReal(), imag+a.getImag());
       // ή, αντί για real και imag: this.real και this.imag
       // ή, αντί για a.getReal() και a.getImag(): a.real και a.imag
  }

  // Η add(Complex a) θα μπορούσε να είναι και ως εξής, κάνοντας
  // χρήση της add(Complex a, Complex b) που ακολουθεί παρακάτω:
  //
  // public Complex add(Complex a)
  // {
  //   return add(this, a); // Το this αναφέρεται στο  αντικείμενο
  //                        // με το οποίο καλείται η μέθοδος
  // }

public static Complex add(Complex a, Complex b)
{
  return new Complex(a.getReal()+b.getReal(),a.getImag()+b.getImag());
}
```

Η πρώτη μέθοδος (`public Complex add(Complex a)`) δέχεται ένα αντικείμενο τύπου `Complex` και δημιουργεί ένα νέο αντικείμενο της `Complex` (το οποίο και επιστρέφει) με τιμές των μεταβλητών του τα αθροίσματα των τιμών του αντικειμένου που δέχεται (`a`) με τις τιμές του αντικειμένου με το οποίο καλεί κάποιος τη μέθοδο (μεταβλητές `real` και `imag`). *[Παράδειγμα κλήσης ακολουθεί παρακάτω]*

Η δεύτερη μέθοδος (`public static Complex add(Complex a, Complex b)`) ακολουθεί διαφορετική προσέγγιση στην πρόσθεση δύο αντικειμένων. Δέχεται και τα δύο αντικείμενα ως παραμέτρους εισόδου και επιστρέφει νέο αντικείμενο με τιμές μεταβλητών τα αθροίσματα των τιμών των δύο αντικειμένων που δέχεται. Σε αυτή την περίπτωση, δεν έχει νόημα η μέθοδος να καλείται για κάποιο συγκεκριμένο αντικείμενο (αφού και τα δύο αντικείμενα που "την ενδιαφέρουν" τα δέχεται ως ορίσματα), γι αυτό και δηλώνεται ως `static`, ώστε να καλείται γενικά.

Η δεύτερη εκδοχή της πρώτης μεθόδου (που φαίνεται σε σχόλιο στον παραπάνω κώδικα), στην ουσία χρησιμοποιεί τη δεύτερη μέθοδο (με τις δύο παραμέτρους εισόδου), στην οποία στέλνει ως πρώτο όρισμα το αντικείμενο με το οποίο κάποιος την καλεί (αυτό αντιπροσωπεύει η λέξη `this`) και ως δεύτερο όρισμα το δικό της όρισμα (`a`).

Με αντίστοιχο τρόπο θα μπορούσαν να προστεθούν στην κλάση `Complex` και άλλες μέθοδοι, για τις υπόλοιπες πράξεις μιγαδικών αριθμών.

Η κλάση εφαρμογής ενός προγράμματος που χρησιμοποιεί τη βελτιωμένη έκδοση της κλάσης `Complex` θα μπορούσε να είναι η ακόλουθη:

```
import java.util.*;

public class TestComplex
{
  public static void main(String [] args)
  {
    Scanner input = new Scanner(System.in);
    System.out.print("Δώσε το πραγματικό και το φανταστικό μέρος του πρώτου μιγαδικού: ");
    double r1 = input.nextDouble();
    double i1 = input.nextDouble();
    System.out.print("Δώσε το πραγματικό και το φανταστικό μέρος του δεύτερου μιγαδικού: ");
    double r2 = input.nextDouble();
    double i2 = input.nextDouble();

    Complex c1 = new Complex(r1,i1);
    Complex c2 = new Complex(r2,i2);

    System.out.print("Ο πρώτος μιγαδικός αριθμός είναι ο: ");
    System.out.println(c1.printComplex());
    System.out.print("Ο δεύτερος μιγαδικός αριθμός είναι ο: ");
    System.out.println(c2.printComplex());

    Complex c3 = c1.add(c2);
    // ή εναλλακτικά, με κλήση της static εκδοχής της μεθόδου add:
    // Complex c3 = Complex.add(c1,c2);

    System.out.print("Το άθροισμα των δύο μιγαδικών ισούται με: ");
    System.out.println(c3.printComplex());
  }
}
```

Ένα παράδειγμα εκτέλεσης του προγράμματος:

```
Δώσε το πραγματικό και το φανταστικό μέρος του πρώτου μιγαδικού: 3 7
Δώσε το πραγματικό και το φανταστικό μέρος του δεύτερου μιγαδικού: 2 -1
Ο πρώτος μιγαδικός αριθμός είναι ο: 3.0 + 7.0 i
Ο δεύτερος μιγαδικός αριθμός είναι ο: 2.0 - 1.0 i
Το άθροισμα των δύο μιγαδικών ισούται με: 5.0 + 6.0 i
```

Overloading μεθόδων (υπερφόρτωση)

Στο προηγούμενο παράδειγμα της κλάσης `Complex`, στη μέθοδο `add`, παρατηρούμε το φαινόμενο της *υπερφόρτωσης μεθόδων*, σε αντιστοιχία με την υπερφόρτωση κατασκευαστών που αναφέρθηκε νωρίτερα. Όταν σε μία κλάση υπάρχουν μέθοδοι *με το ίδιο όνομα* αλλά με διαφορετικό πλήθος παραμέτρων εισόδου ή με διαφορετικούς τύπους παραμέτρων εισόδου, παρατηρείται το φαινόμενο της υπερφόρτωσης μεθόδου (overloading). Ο τύπος της μεθόδου δεν είναι από μόνος του ικανό στοιχείο διαφοροποίησης. Π.χ., στο παρακάτω παράδειγμα παρατηρείται υπερφόρτωση της μεθόδου `myMethod`:

```
public class MyClass
{
  public int myMethod(int x)
  {
     return 2*x;
  }

  public int myMethod(int x, int y)
  {
     return x*y;
  }
}
```

ενώ το παρακάτω παράδειγμα είναι λάθος:

```
public class MyClass
{
  public int myMethod(int x)
  {
     return 2*x;
  }

  public double myMethod(int x)  // ΣΦΑΛΜΑ!
  {
     return Math.sqrt(x);
  }
}
```

Παράδειγμα (πρόγραμμα με δύο κλάσεις υποστήριξης):

Μια κλάση `Point` για σημεία στο επίπεδο:

```
public class Point
{
    private double x, y;

    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public Point()
    {
        x = 0;
        y = 0;
    }

    // Υπολογισμός της απόστασης σημείων (π.χ. σε m)
    public double distance(Point pt)
    {
        return Math.sqrt((x-pt.x)*(x-pt.x) + (y-pt.y)*(y-pt.y));
    }

    public String printPoint()
    {
        return "(" + x + "," + y + ")";
    }
}
```

Μια κλάση `Charge` για φορτία ηλεκτροστατικών πεδίων Coulomb:

```
public class Charge
{

    private double q;  // τιμή φορτίου (σε Coulomb (C))
    private Point pt;   // θέση φορτίου

    private final double k = 8.99e9;  // σταθερά Coulomb (N m^2 / C^2)

    public Charge(double q, Point t)
    {
        this.q = q;
        pt = t;
    }

    public Charge(double q, double x, double y)
    {
        this.q = q;
        pt = new Point(x, y);
    }

    // Υπολογισμός δυναμικού που δημιουργείται από το φορτίο σε σημείο (t),
    // σε Volt (V)
    public double potentialAt(Point t)
    {
        return k*q/pt.distance(t);
    }

    // Υπολογισμός δυναμικού που δημιουργείται από το φορτίο σε σημείο (x,y) (V)
    public double potentialAt(double x, double y)    // Υπερφόρτωση
    {
        return k*q/pt.distance(new Point(x, y));
    }

    public String printCharge()
    {
        return q + " C στο σημείο " + pt.printPoint();
    }

}
```

Η λέξη `final` στον ορισμό μιας μεταβλητής την κάνει **σταθερά**, δηλαδή η μεταβλητή αυτή δεν μπορεί να αλλάξει τιμή μετά την αρχικοποίησή της.

Η μέθοδος εφαρμογής του προγράμματος:

```
public class RunCharge
{
  public static void main(String [] args)
  {
    Point t = new Point(3.0, 4.0);  // θέση φορτίου
    Point p = new Point(4.0, 5.0);  // θέση σημείου
    Charge c = new Charge(100, t);  // δημιουργία φορτίου στη θέση t

    System.out.println(t.printPoint());  // εκτύπωση συντεταγμένων του t
    System.out.println(p.printPoint());  // εκτύπωση συντεταγμένων του p

    System.out.println(c.printCharge()); // εκτύπωση φορτίου (τιμή / θέση)
    System.out.println(c.potentialAt(p) + "V");// εκτύπωση δυναμικού στο σημείο p
    System.out.println(c.potentialAt(4.0, 4.0) + "V"); // δυναμικό στο σημείο (4,4)
  }

}
```

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 6

Pass-by-value και φαινομενικό pass-by-reference

Γενικά, στις γλώσσες προγραμματισμού, οι όροι "pass-by-value" (πέρασμα με τιμή) και "pass-by-reference" (πέρασμα με αναφορά) αναφέρονται στον τρόπο με τον οποίο μεταβιβάζει (περνάει) μία μέθοδος (που καλεί κάποια άλλη) τα ορίσματα εισόδου στην καλούμενη μέθοδο. Ένα τέτοιο πέρασμα (pass) μπορεί να γίνει είτε περνώντας *τις τιμές* των ορισμάτων αυτών στην καλούμενη μέθοδο (pass-by-value), είτε περνώντας *αναφορές*[1] των ορισμάτων αυτών (pass-by-reference).

Ουσιαστικά, στην πρώτη περίπτωση μεταβιβάζεται στην καλούμενη μέθοδο ένα *αντίγραφο* του ορίσματος, ενώ στη δεύτερη περίπτωση μεταβιβάζεται το ίδιο το όρισμα (μέσω της μεταβίβασης της αναφοράς σε αυτό). Έτσι, στην περίπτωση περάσματος με τιμή, οποιαδήποτε αλλαγή γίνει στο μεταβιβαζόμενο όρισμα, δεν επηρεάζει τη μεταβλητή που μεταβιβάστηκε στη μέθοδο κλήσης (αφού η όποια αλλαγή έγινε σε αντίγραφό της και όχι στην ίδια τη μεταβλητή), ενώ στην περίπτωση περάσματος με αναφορά, οποιαδήποτε αλλαγή γίνει στο μεταβιβαζόμενο όρισμα επηρεάζει και τη μεταβλητή (αντικείμενο) στη μέθοδο κλήσης (αφού έχει μεταβιβαστεί το ίδιο το όρισμα μέσω της αναφοράς του, και όχι ένα αντίγραφό του).

Ας δούμε τι ακριβώς συμβαίνει στη Java.

Έστω μια κλάση υποστήριξης `ClassA`:

```
public class ClassA
{
   private int x, y;
   public ClassA(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   public void setValues(int a, int b)
   {
      x = a;  y = b;
   }

   public int getX()
   {
      return x;
   }
   public int getY()
   {
      return y;
   }
}
```

1 Αναφορά (reference) γενικά στις γλώσσες προγραμματισμού ονομάζεται μια μεταβλητή που αποτελεί ψευδώνυμο ενός αντικειμένου, δηλαδή στην ουσία ένα δεύτερο όνομα για το αντικείμενο. Όπως θα αναλυθεί παρακάτω (και στη 2η παραπομπή), η Java δεν ακολουθεί ακριβώς αυτή την ορολογία, με αποτέλεσμα να δημιουργούνται διάφορες συγχύσεις ως προς τον τρόπο μεταβίβασης των ορισμάτων της όταν αυτά είναι αντικείμενα.

και η κλάση εφαρμογής `Program` που περιέχει τη `main` και μία μέθοδο `add5`:

```
public class Program
{
   public static void main (String[] args)
   {
      ClassA obj = new ClassA(5,10);
      int z = 100;
      System.out.println("Τιμές των x, y και z πριν την κλήση της μεθόδου:");
      System.out.println("x = " + obj.getX());
      System.out.println("y = " + obj.getY());
      System.out.println("z = " + z);

      add5(obj,z);   // Κλήση της add5 που υλοποιείται παρακάτω
      System.out.println("Τιμές των x, y και z μετά την κλήση της μεθόδου:");
      System.out.println("x = " + obj.getX());
      System.out.println("y = " + obj.getY());
      System.out.println("z = " + z);
   }

   public static void add5(ClassA a, int b)
   {
      a.setValues(a.getX()+5, a.getY()+5);
      b+=5;
   }
}
```

Το πρόγραμμα αυτό θα εκτυπώσει τα εξής:

```
Τιμές των x, y και z πριν την κλήση της μεθόδου:
x = 5
y = 10
z = 100
Τιμές των x, y και z μετά την κλήση της μεθόδου:
x = 10
y = 15
z = 100
```

Δηλαδή, η τιμή της μεταβλητής `z` δεν άλλαξε μετά την κλήση της μεθόδου `add5`, ενώ αντιθέτως οι τιμές των μεταβλητών `x` και `y` του αντικειμένου `obj` της κλάσης `ClassA` άλλαξαν.

Στη Java όλες οι παράμετροι περνιούνται με τιμή (pass-by-value) από τη μέθοδο που κάνει την κλήση στη μέθοδο που καλείται. Αυτό ισχύει *και* για τις πρωτογενείς μεταβλητές *και* για τα αντικείμενα. Δηλαδή, και τα αντικείμενα περνιούνται με τιμή (pass-by-value). Θεωρητικά στη Java δεν υπάρχει πέρασμα με αναφορά (pass-by-reference) όπως υπάρχει σε άλλες γλώσσες προγραμματισμού.

Τότε γιατί στο παραπάνω παράδειγμα φαίνεται να γίνεται pass-by-reference στην περίπτωση του αντικειμένου; Αυτό συμβαίνει γιατί στη Java το κάθε αντικείμενο ουσιαστικά είναι μια αναφορά στη θέση μνήμης που περιέχονται τα στοιχεία (δεδομένα) του αντικειμένου, δηλαδή είναι ένας δείκτης (pointer), κατά την ορολογία άλλων γλωσσών προγραμματισμού[2]. Κατά τη μεταβίβαση ενός αντικειμένου, μπορούν να γίνουν αλλαγές στα δεδομένα στα οποία "δείχνει" η αναφορά του

2 Η λέξη "αναφορά" (reference) στη Java δεν χρησιμοποιήθηκε σωστά κατά τη δημιουργία της γλώσσας. Ουσιαστικά οι "αναφορές" στη Java είναι οι "δείκτες" (pointers) άλλων γλωσσών προγραμματισμού, δηλαδή μεταβλητές που οι τιμές τους είναι οι θέσεις μνήμης αντικειμένων. Αυτός είναι και ο λόγος που δημιουργείται όλη αυτή η σύγχυση για τον τρόπο μεταβίβασης ορισμάτων στη Java. Οι δείκτες μεταβιβάζονται με τιμή, αλλά ονομάζονται αναφορές, οπότε δημιουργείται σύγχυση με τη "μεταβίβαση με αναφορά" άλλων γλωσσών προγραμματισμού.

αντικειμένου, αλλά δεν μπορεί να αλλάξει το πού "δείχνει" η αναφορά του αντικειμένου (η οποία συνεχίζει να "δείχνει" πάντα στο ίδιο αντικείμενο).

Για να κατανοήσει κανείς τι ακριβώς συμβαίνει με τα αντικείμενα, πρέπει καταρχάς να κατανοήσει τι συμβαίνει όταν μια απλή μεταβλητή περνιέται με τιμή κατά την κλήση μιας μεθόδου. Όταν καλείται μια μέθοδος που δέχεται μια παράμετρο εισόδου, η μεταβλητή που δίνεται από τη μέθοδο που κάνει την κλήση στη θέση της παραμέτρου (δηλαδή το όρισμα), αντιγράφεται σε μία νέα προσωρινή μεταβλητή που θα χρησιμοποιηθεί ως παράμετρος για την κλήση. Το ίδιο συμβαίνει και στην περίπτωση που αντί για μεταβλητή περνιέται ένα αντικείμενο ως παράμετρος εισόδου στην καλούμενη μέθοδο: το αντικείμενο αντιγράφεται σε ένα προσωρινό αντικείμενο που αποστέλλεται στη μέθοδο που καλείται. *Όμως, αυτό το προσωρινό αντικείμενο δεν είναι τίποτε άλλο από μια διαφορετική αναφορά (δείκτης) στο ίδιο αρχικό αντικείμενο* (αφού τα αντικείμενα είναι αναφορές στις θέσεις μνήμης που περιέχονται τα αντικείμενα, δηλαδή δείκτες στα δεδομένα τους). Επομένως, ό,τι αλλαγές γίνουν στα δεδομένα του αντικειμένου από την κληθείσα μέθοδο, θα ισχύουν και για το αντικείμενο που περάστηκε ως όρισμα στην καλούσα μέθοδο, αφού πρόκειται για το ίδιο αντικείμενο.

Για αυτόν τον λόγο, στο παραπάνω παράδειγμα οι μεταβλητές του αντικειμένου `obj` παραμένουν αλλαγμένες μετά την επαναφορά της ροής του κώδικα από τη μέθοδο `add5` στη μέθοδο `main` (μετά την κλήση της μεθόδου `add5`), σε αντίθεση με την απλή μεταβλητή `z`, η οποία παραμένει αμετάβλητη στην αρχική της τιμή.

Σημείωση: *Επειδή στη Java οι* ***πίνακες*** *(που θα αναφερθούν στο επόμενο κεφάλαιο) είναι στην ουσία αντικείμενα, ό,τι ισχύει για τα αντικείμενα όσον αφορά το πέρασμα των τιμών τους, ισχύει και για αυτούς.*

Αν μέσα στη μέθοδο `add5` ή σε κάποια άλλη αντίστοιχη μέθοδο δεν άλλαζαν οι τιμές μεταβλητών του αντικειμένου `obj`, αλλά άλλαζε *ολόκληρο* το αντικείμενο (σε αντιστοιχία με την αλλαγή μιας απλής μεταβλητής), τότε η αλλαγή αυτή θα έμενε στη μέθοδο και δε θα "πέρναγε" στη μέθοδο `main`, δηλαδή θα συνέβαινε ό,τι ακριβώς συμβαίνει και με τις πρωτογενείς μεταβλητές. Αυτό είναι μια "απόδειξη" ότι ***και*** *τα αντικείμενα ουσιαστικά γίνονται pass-by-value στη Java*. Π.χ., αν στο παραπάνω παράδειγμα, στην κλάση `Program` υπήρχε, αντί της μεθόδου `add5`, η μέθοδος `change`, οπότε η `Program` γινόταν ως εξής:

```
public class Program
{
   public static void main (String[] args)
   {
      ClassA obj = new ClassA(5,10);
      int z = 100;
      System.out.println("Τιμές των x, y και z πριν την κλήση της μεθόδου:");
      System.out.println("x = " + obj.getX());
      System.out.println("y = " + obj.getY());
      System.out.println("z = " + z);

      change(obj,z);    // Κλήση της change που ορίζεται παρακάτω

      System.out.println("Τιμές των x, y και z μετά την κλήση της μεθόδου:");
      System.out.println("x = " + obj.getX());
      System.out.println("y = " + obj.getY());
      System.out.println("z = " + z);
   }
```

```
   public static void change(ClassA a, int b)
   {
      a = new ClassA(50,55);
      b+=5;
   }
}
```

τότε το πρόγραμμα θα εκτύπωνε τα εξής:

```
Τιμές των x, y και z πριν την κλήση της μεθόδου:
x = 5
y = 10
z = 100
Τιμές των x, y και z μετά την κλήση της μεθόδου:
x = 5
y = 10
z = 100
```

Πλέον η συμπεριφορά των τιμών των `x` και `y` του αντικειμένου `obj` είναι η ίδια με αυτή μιας κανονικής μεταβλητής, δηλαδή παραμένουν στις προηγούμενες τιμές τους (5 και 10 αντίστοιχα), πράγμα που σημαίνει ότι όντως το αντικείμενο `obj` περάστηκε με τιμή (pass-by-value) στη μέθοδο `change` και δε μεταβλήθηκε από αυτή. Το μόνο που άλλαξε σε σχέση με πριν είναι ότι πλέον η μέθοδος δεν τροποποιεί μεμονωμένα τις τιμές των μεταβλητών του αντικειμένου που δέχεται, αλλά τροποποιεί "ολόκληρο" το αντικείμενο, άρα το μόνο που τροποποιεί είναι την αναφορά στη θέση μνήμης του προσωρινού αντικειμένου που δημιουργείται, το οποίο τώρα απλά αναφέρεται στη θέση μνήμης ενός νέου αντικειμένου (με τιμές μεταβλητών 50 και 55). Οι μεταβλητές του αρχικού αντικειμένου `obj` φυσικά παραμένουν αμετάβλητες μέσα στη μέθοδο `main`, όπως ακριβώς συμβαίνει και με τη μεταβλητή `z`.

Όλο αυτό μπορεί να γίνει καλύτερα κατανοητό εάν γίνει σαφές ότι μια δήλωση της μορφής:

```
ClassA a;
```

δε δημιουργεί αντικείμενο της κλάσης `ClassA`, δημιουργεί απλά έναν "δείκτη" (μια "αναφορά") για ένα αντικείμενο της κλάσης `ClassA`. Άρα, όταν καλείται η μέθοδος `change` στο παραπάνω παράδειγμα, αρχικά δημιουργείται μια αναφορά με όνομα `a` για ένα αντικείμενο της `ClassA`, και στη συνέχεια, με την εντολή `a = new ClassA(50,55);` δημιουργείται το αντικείμενο στο οποίο δείχνει η αναφορά με όνομα `a`. Αν δεν υπάρχει κάποια τέτοια δημιουργία αντικειμένου μέσα στη μέθοδο και απλά κληθούν μέθοδοι του αντικειμένου της "αναφοράς" `a` (όπως στη μέθοδο `add5` του αρχικού παραδείγματος), τότε καλούνται με το αντικείμενο-όρισμα που στέλνεται στη μέθοδο (`obj`), και αν μεταβάλουν κάποια πεδία τού αντικειμένου αυτού, τότε οι όποιες μεταβολές παραμένουν και μετά την ολοκλήρωση εκτέλεσης της μεθόδου.

Η Κλάση `String`

Κάθε συμβολοσειρά (ή αλφαριθμητικό) (string) είναι αντικείμενο της κλάσης `String`.

Βασική ιδιότητα: Τα string, δηλ. τα αντικείμενα της κλάσης `String`, δεν τροποποιούνται.

Τρόποι δημιουργίας συμβολοσειρών:

- Με τη χρήση εισαγωγικών: `String str = "Hello ";`
- Με τη χρήση των + ή += πάνω σε υπάρχοντα strings
  `"Hello " + "world"` → νέο string → `Hello world`
- Με κανονική δημιουργία αντικειμένου μέσω της `new`.

Η `String` έχει δύο κατασκευαστές (constructors):

`public String()` και `public String(String value)`

π.χ. `String str1 = new String();`
`String str2 = new String("Hello");`

Προσοχή στον τελεστή + :

```
double x=15, y=25;
System.out.println(x+y);
System.out.println("Το άθροισμα είναι " + x + y);
System.out.println("Το άθροισμα είναι " + (x + y));
```

Ο κώδικας αυτός θα εκτυπώσει:

```
40.0
Το άθροισμα είναι 15.025.0
Το άθροισμα είναι 40.0
```

Δηλαδή, όταν ο τελεστής + βρει κάποιο string, προσθέτει απλά τις τιμές των μεταβλητών που βρίσκει μετά σαν συνέχεια στο string αυτό. Άρα, για να γίνει η πράξη της πρόσθεσης των δύο `double` μεταβλητών, θα πρέπει να μπει σε παρένθεση για να προηγηθεί της συνένωσης με το string.

ΠΡΟΣΟΧΗ: Ο πρώτος τρόπος δημιουργίας συμβολοσειρών, δηλαδή η εντολή της μορφής:

```
String όνομα_συμβολοσειράς = "τιμή";
```

έχει την πολύ σημαντική ιδιότητα να αποθηκεύει στον ίδιο χώρο μνήμης αντικείμενα της `String` με κοινή τιμή. Δηλαδή, αν:

```
String s1 = "Hello";
String s2 = "Hello";
```

τα `s1` και `s2` αναφέρονται στο *ίδιο* αντικείμενο (δηλαδή υπάρχει *ένα* αντίγραφο της λέξης `"Hello"` στη μνήμη). Η ιδιότητα αυτή είναι πολύ σημαντική ως προς τη διαχείριση μνήμης ενός προγράμματος και είναι και ο λόγος για τον οποίο τα αντικείμενα τύπου `String` είναι μη τροποποιήσημα (immutable). Αν μπορούσαν να τροποποιηθούν, τότε στο παραπάνω παράδειγμα, μια τροποποίηση π.χ. στο `s2` θα τροποποιούσε και το `s1` (όπως συμβαίνει και στα κανονικά αντικείμενα). Αυτό όμως θα μπορούσε να δημιουργήσει μεγάλη σύγχυση σε κάποιο πρόγραμμα.

ΠΡΟΣΟΧΗ: Αν δημιουργηθεί και ένα τρίτο string `s3` με το περιεχόμενο `"Hello"` αλλά με τη χρήση του `new`:

```
String s3 = new String("Hello");
```

τότε αυτό είναι ένα ξεχωριστό αντικείμενο στη μνήμη, παρόλο που έχει το ίδιο περιεχόμενο. Άρα, η ιδιότητα χρήσης κοινής μνήμης για συμβολοσειρές ίδιας τιμής ισχύει μόνο για αυτές που έχουν δημιουργηθεί με τον πρώτο τρόπο δημιουργίας συμβολοσειρών, ο οποίος μοιάζει με αυτόν της δημιουργίας πρωτογενών μεταβλητών (χωρίς τη χρήση της `new`).

Η ευκολία αυτή δημιουργίας string χωρίς τη χρήση της `new` και η ιδιότητα της μη-δημιουργίας νέων αντικειμένων σε περίπτωση ίδιου περιεχομένου (ίδιας συμβολοσειράς), κάνει τη χρήση της κλάσης `String` πολύ ελκυστική (σε αντιδιαστολή με άλλες κλάσεις συμβολοσειρών όπως η `StringBuffer` και η `StringBuilder`, τα αντικείμενα των οποίων είναι τροποποιήσιμα).

<u>Βασικές μέθοδοι της κλάσης `String`</u>

- `public int length()` → επιστρέφει το πλήθος των χαρακτήρων του αλφ/κού

π.χ.
```
String a = "blah blah";
int b = a.length();   // → b = 9
```

- `public char charAt(int index)` → επιστρέφει τον χαρακτήρα στη θέση `index`

```
char d = a.charAt(3);  // → d = 'h'   (μετράμε από το 0!)
```

- `public int indexOf(char ch)` → επιστρέφει την πρώτη θέση του `ch`

```
a.indexOf('a');  → 2
```

- `public int indexOf(char ch, int start)` → επιστρέφει την πρώτη θέση του `ch` από τη θέση `start` και μετά.

```
a.indexOf('a',3);  → 7
```

- `public int indexOf(String str)` → επιστρέφει την πρώτη θέση του `str`

- `public int indexOf(String str, int start)` → επιστρέφει την πρώτη θέση του `str` από τη θέση `start` και μετά.

- `public int lastIndexOf(char ch)` → επιστρέφει την τελευταία θέση του `ch`

- `public int lastIndexOf(char ch, int start)` → επιστρέφει την τελευταία μέχρι και το `start` θέση του `ch`

- `...` [αντίστοιχα και για παράμετρο `String str`].

Όλες αυτές οι μέθοδοι αν δεν βρουν αυτό που ψάχνουν στο string (το `ch` ή το `str`), επιστρέφουν **`-1`**.

Μέθοδοι σύγκρισης

- `public boolean equals(String str)`
  - → `true`, αν βρει ίδιο μήκος και ακριβώς ίδιους χαρακτήρες
  - → `false`, σε διαφορετική περίπτωση

π.χ.
```
String e = "blah blah";
boolean f = a.equals(e);  // → f = true
```

Ο τελεστής == όταν εφαρμοστεί σε δύο string συγκρίνει τις θέσεις μνήμης τους και όχι το περιεχόμενό τους. Προσοχή στο ότι αν δύο string έχουν δημιουργηθεί χωρίς τη χρήση του `new` και έχουν το ίδιο περιεχόμενο, τότε η σύγκρισή τους με το == θα επιστρέψει `true`, αφού όπως προαναφέρθηκε, τέτοια αντικείμενα ίδιου περιεχομένου είναι ουσιαστικά ένα κοινό αντικείμενο.

- `public boolean equalsIgnoreCase(String str)`
  Αγνοεί κεφαλαία ή πεζά γράμματα

- `public int compareTo(String str)`
  - → `< 0` αν κάποιο string έχει < μήκος του `str`
  - → `= 0` αν κάποιο string έχει = μήκος με το `str`
  - → `> 0` αν κάποιο string έχει > μήκος του `str`

```
String h = "blah";
int i = h.compareTo(a);  // → i<0
```

- `public boolean regionMatches(int start, String str, int strStart, int len)`

```
h.regionMatches(1,a,6,3);  // → true
```

```
h →  b l a h
     0 1 2 3

a →  b l a h  b l a h
     0 1 2 3 4 5 6 7 8
```

```
boolean b = a.regionMatches(2,"Ah",0,2);  // → false  (ah ≠ Ah)
```

- `public boolean regionMatches(boolean ignoreCase, int start, String str, int strStart, int len)`

```
boolean b = a.regionMatches(true,2,"Ah",0,2);  // → true
```

Μέθοδοι ελέγχου αρχής/τέλους:

- `public boolean startsWith(String prefix)`

- `public boolean endsWith(String suffix)`

```
boolean b = a.endsWith("ah")  → true
```

Μέθοδοι δημιουργίας νέων αντικειμένων `String`

- `public String replace(char oldChar, char newChar)`

```
String j = a.replace('a','i');  // → blih blih
```

Δεν αλλάζει το string `a`. Φτιάχνει **νέο** string (αντικείμενο)

- `public String toLowerCase()`

Παράδειγμα: Στην περίπτωση απλών αντικειμένων π.χ. μιας κλάσης υποστήριξης `SuppClass`, θα μπορούσαμε να έχουμε τον εξής κώδικα σε μια άλλη κλάση:

```
// Δημιουργία αντικειμένου της SuppClass
SuppClass obj1 = new SuppClass();

// Απόδοση τιμής στη μεταβλητή x του obj1
obj1.x = 10;

// Δημιουργία αντίγραφου του αντικειμένου obj1
SuppClass obj2 = obj1;

// Απόδοση τιμής στη μεταβλητή x του obj2
obj2.x = 5;

// Εκτύπωση στην οθόνη του x του obj1
System.out.println(obj1.x);
```

Ο κώδικας αυτός θα εκτυπώσει την τιμή **5**.

Στην ουσία έχουμε **ένα** αντικείμενο της `SuppClass`, το οποίο έχει δύο ονόματα, `obj1` και `obj2`. Οπότε, αλλάζοντας την τιμή του `x` στο `obj2`, στην ουσία αλλάζει η τιμή και στο `obj1`.

Αν τώρα έχουμε έναν αντίστοιχο κώδικα, αλλά με αντικείμενα της κλάσης `String` αντί κάποιας άλλης κλάσης:

```
// Δημιουργία αντικειμένου «τύπου» String
String k = "Hello";
```

```
// Δημιουργία αντίγραφου του αντικειμένου k
String m = k;

// «Τροποποίηση» του αντικειμένου m
m = m.toLowerCase();

// Εκτύπωση στην οθόνη της τιμής του αντικειμένου k
System.out.println(k);
```

Ο κώδικας αυτός θα εκτυπώσει **`Hello`** και όχι `hello` όπως θα περίμενε κάποιος, σε αντιστοιχία με το προηγούμενο παράδειγμα με την `SuppClass`.

Αυτό συμβαίνει γιατί κατά την εκτέλεση της εντολής `m.toLowerCase()` δεν αλλάζει η τιμή του `m` και άρα και του `k`, αλλά δημιουργείται ένα **νέο** αντικείμενο, στο οποίο «δείχνει» πλέον το αντικείμενο `m`. **Άρα, το αρχικό αντικείμενο `k` μένει αμετάβλητο (`Hello`), το `m` παύει να υπάρχει ως αντίγραφο του `k` και δημιουργείται ένα νέο αντικείμενο (`hello`) με το όνομα `m`.**

Τα αντικείμενα τύπου `String` λοιπόν, δεν μπορεί να μεταβληθούν. Κάθε «τροποποίηση» επιστρέφει ένα <u>νέο αντικείμενο</u> τύπου `String`.

- `public String toUpperCase()`

- `public String trim()` → κόβει τα κενά σε αρχή και τέλος

- `public String concat(String str)` → το ίδιο με το +

```
        // Apo prin: a=blah blah kai h=blah
    String s = a.concat(h); // → blah blahblah  (ίδιο με το a+h;)
```

<u>Μετατροπές από/σε `String`</u>

Από μεταβλητή τύπου `boolean`, `int`, `long`, `float`, `double`, `char` σε `String`:

```
        String.valueOf(<μεταβλητή>);
```

π.χ.,

```
    int n = 10;
    String p = String.valueOf(n);  // → p = "10"
```

Από `String` σε `boolean` → `Boolean.parseBoolean(str)`
σε `int` → `Integer.parseInt(str)`
σε `long` → `Long.parseLong(str)`
σε `float` → `new Float(str).floatValue()`
σε `double` → `Double.parseDouble(str)`

Ειδικοί χαρακτήρες

Ο χαρακτήρας \ χρησιμοποιείται σε ειδικές περιπτώσεις, όπως για την εκτύπωση των εισαγωγικών:

```
System.out.println("Say \"Hi\"!"); → τυπώνει: Say "Hi"!
```

(το παραπάνω string έχει 9 χαρακτήρες και όχι 11. Το \ δεν μετράει ως χαρακτήρας)

Για εκτύπωση του χαρακτήρα \ χρησιμοποιείται το: \\

```
System.out.println("abc\\def"); → τυπώνει: abc\def
```

\n → νέα γραμμή (ENTER)
\t → tab

Η Κλάση `StringBuffer`

Είναι μια κλάση για τη δημιουργία συμβολοσειρών που μπορούν να τροποποιηθούν (σε αντίθεση με τις συμβολοσειρές της κλάσης `String` που είναι μη-τροποποιήσημες).

Οι βασικοί της κατασκευαστές είναι οι εξής:

- `public StringBuffer()` → δημιουργία κενής συμβολοσειράς 16 θέσεων
- `public StringBuffer(String str)` → δημιουργία τροποποιήσιμης συμβολοσειράς με αρχική τιμή την τιμή `str`
- `public StringBuffer(int capacity)` → δημιουργία κενής συμβολοσειράς δεδομένου αριθμού θέσεων (capacity)

Στην ουσία τα αντικείμενα της `StringBuffer` είναι *διανύσματα χαρακτήρων* μεταβλητού μεγέθους.

Μια μέθοδος που χρησιμοποιείται συχνά για τροποποίηση μιας συμβολοσειράς (όχι δημιουργία καινούριας), είναι η `setCharAt`:

```
public void setCharAt(int index, char newChar)
```

η οποία αντικαθιστά τον υπάρχοντα χαρακτήρα στη θέση `index` μιας συμβολοσειράς με τον χαρακτήρα `newChar`.

Η βασικότερη μέθοδος της κλάσης `StringBuffer` είναι η μέθοδος `append` η οποία υπάρχει σε πολλές *υπερφορτωμένες* μορφές. Η κύρια μορφή της είναι αυτή της συνένωσης συμβολοσειρών:

```
public void append(String str)
```

η οποία προσθέτει τη συμβολοσειρά `str` σε κάποιο συγκεκριμένο `StringBuffer`, η οποία μπορεί να κληθεί και επαναλαμβανόμενα, όπως φαίνεται στο παρακάτω παράδειγμα.

Π.χ., έστω ότι έχουν δηλωθεί και έχουν πάρει τιμές τρεις μεταβλητές τύπου `String`, οι: `title`, `firstName` και `lastName`. Εάν προσπαθούσε κάποιος να ενώσει αυτές τις τρεις συμβολοσειρές συνενώνοντας `String` και αποθηκεύοντας το αποτέλεσμα σε ένα νέο `String` με την εντολή:

```
String name = title + " " + firstName + " " + lastName;
```

τότε θα είχε δημιουργήσει 4 ενδιάμεσα `String` μέχρι να επιτευχθεί η τελική συμβολοσειρά: Αρχικά το `title + " "`, μετά το `title + " " + firstName`, κοκ. Αντιθέτως, με τη χρήση της μεθόδου `append` της `StringBuffer`, η συνένωση των συμβολοσειρών θα γινόταν με τροποποίηση μίας μόνο συμβολοσειράς:

```
StringBuffer name = new StringBuffer().append(title).append(" ").
                    append(firstName).append(" ").append(lastName);
```

Μετατροπές `String` / `StringBuffer`

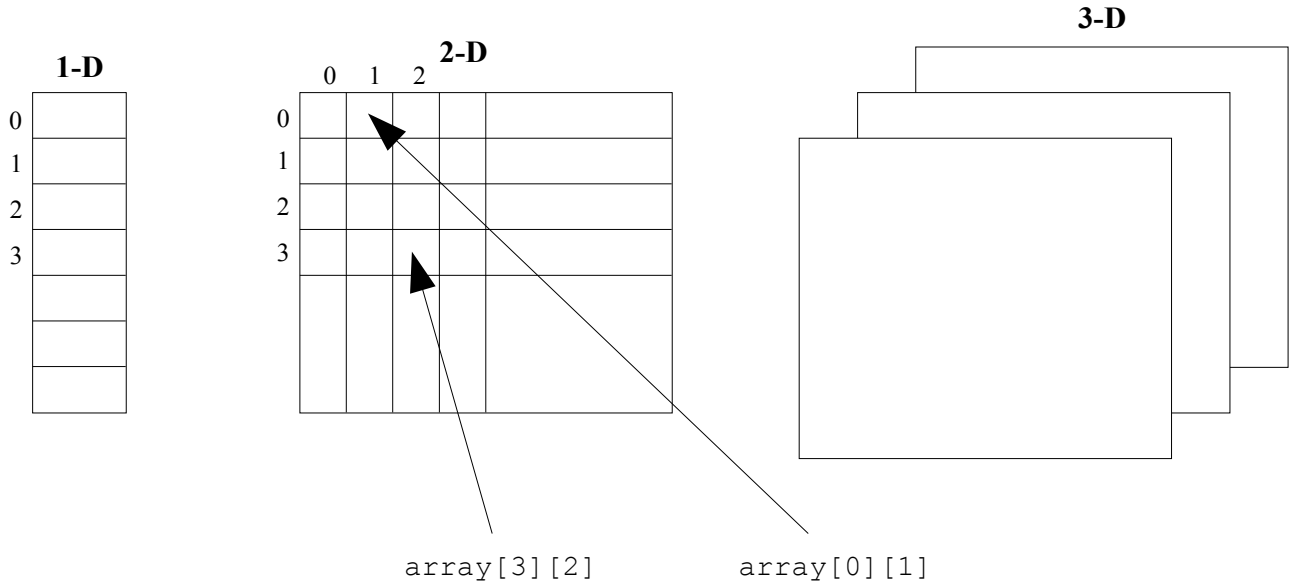
Η μέθοδος `toString` μπορεί να μετατρέψει ένα `StringBuffer` σε `String`. Κάνοντας χρήση της ικανότητας της `StringBuffer` να τροποποιεί συμβολοσειρές και της `toString`, μπορούμε να δημιουργήσουμε μία μέθοδο που δέχεται κάποιο `String` και το επιστρέφει τροποποιημένο:

```
public String changeString(String str)
{
   StringBuffer buffer = new StringBuffer(str);

   // τροποποίηση του buffer
   .
   .
   .

   return buffer.toString();
}
```

Στην ουσία η μέθοδος αυτή κάνει το εξής: δέχεται μια συμβολοσειρά `String`, την αποθηκεύει σε μορφή `StringBuffer`, την τροποποιεί στην επιθυμητή νέα τιμή (άρα, χωρίς τη δημιουργία επιπλέον αντικειμένων), και επιστρέφει την τροποποιημένη συμβολοσειρά σε μορφή `String` πάλι.

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 7

Πίνακες (Arrays)



Γενική δήλωση και δημιουργία μονοδιάστατου πίνακα:

```
τύπος_στοιχείων [ ] όνομα = new τύπος_στοιχείων [μήκος];  // → 1-D
```

και φυσικά μπροστά μπορεί να υπάρχει η ανάλογη ορατότητα (`public`, `private`, κτλ.)

π.χ. `private double [ ] temperature = new double [10];`

Στην ουσία, με αυτόν τον τρόπο δηλώνονται 10 `double` μεταβλητές υπό το ίδιο όνομα, δηλαδή *σαν* να δηλώνονταν:

```
private double temperature[0] = 0;
private double temperature[1] = 0;
  ...
private double temperature[9] = 0;
```

ο τρόπος αυτός δήλωσης μεταβλητών όμως δεν υφίσταται πραγματικά, γι αυτό και η δήλωσή τους γίνεται μέσω της δημιουργίας του αντίστοιχου πίνακα temperature.

Σημαντικά σημεία εδώ είναι ότι:

- Η δημιουργία ενός πίνακα αρχικοποιεί τα στοιχεία του στην τιμή 0 (null, για στοιχεία τύπου char).
- Η αρίθμηση της θέσης των στοιχείων του πίνακα (index) ξεκινάει από την τιμή 0 (και άρα φτάνει μέχρι την τιμή «μήκος–1»)

Δήλωση και δημιουργία δισδιάστατου πίνακα (2-D):

```
τύπος_στοιχείων [ ][ ] όνομα = new τύπος_στοιχείων [size1][size2];
```

Δημιουργία και αρχικοποίηση σε συγκεκριμένες τιμές διαφορετικές του μηδενός:

```
τύπος_στοιχείων [ ] όνομα = {τιμή1, τιμή2, ...};
```

π.χ., `int [ ] daysInMonth = {31, 28, 31, 30, 31, ...};`

ή
```
int [ ] daysInMonth = new int [12];
daysInMonth[0] = 31;
daysInMonth[1] = 28;
daysInMonth[2] = 31;
  ...
daysInMonth[11] = 31;
```

ΠΡΟΣΟΧΗ: Το `daysInMonth[12]` **δεν** υπάρχει. 12 είναι το μέγεθος του πίνακα. Το index μεταβάλλεται από 0 έως 11.

Για 2-D: π.χ. η εντολή:

```
int [ ][ ] a = { {2, 3, 5}, {4, 2, 0}, {0, 2, 1} };
```

δημιουργεί τον πίνακα:

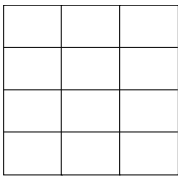
```
        2  3  5
a =     4  2  0
        0  2  1
```

Λεπτομέρεια: Στη Java τυπικά υπάρχουν μόνο μονοδιάστατοι πίνακες! Όταν φτιάχνουμε έναν δισδιάστατο πίνακα, στην ουσία φτιάχνουμε ένα μονοδιάστατο πίνακα για κάθε στοιχείο ενός αρχικού μονοδιάστατο πίνακα. Δηλαδή, όταν φτιάχνουμε ένα δισδιάστατο πίνακα μεγέθους [4]x[3] με την εντολή:

```
int [ ][ ] array1 = new int [4][3];
```

στην αρχή δημιουργείται ένας 1-D πίνακας μεγέθους [4]:

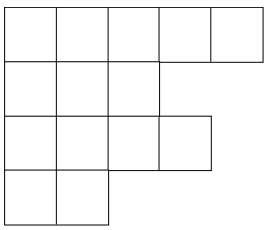
και στη συνέχεια κάθε στοιχείο του γίνεται ένας 1-D πίνακας μεγέθους [3], άρα φαινομενικά δημιουργείται ο 2-D πίνακας διαστάσεων [4] x [3]:



Αυτό δίνει τη δυνατότητα στη Java να δημιουργήσει πίνακες μη-ορθογώνιους, δηλαδή πίνακες με διαφορετικό αριθμό στηλών ανά γραμμή:

```
int [ ][ ] array2 = new int [4][ ];
array2[0] = new int [5];
array2[1] = new int [3];
array2[2] = new int [4];
array2[3] = new int [2];
```

Οι εντολές αυτές δημιουργούν τον ακόλουθο πίνακα:



Πρόσβαση σε στοιχεία πίνακα:

- Για μονοδιάστατους: `όνομα_πίνακα[δείκτης]`
- Για δισδιάστατους: `όνομα_πίνακα[δείκτης_γραμμής][δείκτης_στήλης]`

Η `length` επιστρέφει το μέγεθος ενός πίνακα:

```
double [ ] a = new double [10];
int x = a.length;   // → x=10
```

Οπότε, ένας καλός τρόπος αυτοματοποιημένης αρχικοποίησης ενός πίνακα:
(παράδειγμα):

```
int [ ] b = new int [100];
for (int i=0; i<b.length; i++)
   b[i] = 1;
```

Στην περίπτωση 2-D πινάκων:

```
double [ ][ ] c = new double [10][20];
for (int i=0; i<c.length; i++)
   for (int j=0; j<c[0].length; j++)
      c[i][j] = 1;
```

Το `c[0].length` στην ουσία επιστρέφει το μήκος της πρώτης γραμμής του πίνακα (της γραμμής 0), δηλαδή 20, και το 0 στο `c[0]` θα μπορούσε να είναι οποιοσδήποτε ακέραιος μεταξύ 0 και 9 εφόσον ο `c` είναι ορθογώνιος πίνακας (όλες η γραμμές έχουν το ίδιο πλήθος στοιχείων).

ΕΡΩΤΗΣΗ: Πώς θα μπορούσαμε με παραπλήσιο τρόπο να αρχικοποιήσουμε μη-ορθογώνιο δισδιάστατο πίνακα;

for-each loop (Πρόσβαση σε στοιχεία με χρήση της foreach)

Εκτός από το κλασικό `for`, στις νεώτερες εκδόσεις της Java υπάρχει ένας άλλος τύπος `for`, το οποίο ονομάζεται ***foreach*** και το οποίο στην περίπτωση των πινάκων λειτουργεί όπως παρακάτω, δίνοντας τη δυνατότητα πρόσβασης σε στοιχεία ενός πίνακα *χωρίς* τη χρήση των δεικτών θέσης τους. Η βασική του σύνταξη είναι η εξής:

```
for (τύπος_στοιχείων_πίνακα  μεταβλητή : πίνακας)
   εντολές
```

Έστω ένας μονοδιάστατος πίνακας `arr` πραγματικών (`double`) τιμών:

```
double [] arr = {1.0, 2.0, 3.0, 4.0};
```

Το παρακάτω `for` διατρέχει όλα τα στοιχεία του πίνακα, ένα-προς-ένα:

```
for (double el : arr)
      System.out.println(el);
```

Η συγκεκριμένη εντολή "διαβάζεται": *"Για κάθε στοιχείο el του πίνακα arr..."*.

Η μεταβλητή `el` είναι μια τοπική μεταβλητή του `for`, και πρέπει προφανώς να είναι ίδιου τύπου με τα στοιχεία του πίνακα. Η πρόσβαση στα στοιχεία του πίνακα μέσα στο `for` γίνεται πλέον με το όνομα αυτής της μεταβλητής, χωρίς να απαιτείται η χρήση δεικτών. Στην πρώτη επανάληψη εκχωρείται στην `el` το 1ο στοιχείο του πίνακα, στη 2η το 2ο κοκ. Η χρήση του foreach είναι συνήθως πιο σύντομη, αλλά εμπεριέχει διάφορους περιορισμούς που προκύπτουν από το ότι τα στοιχεία διατρέχονται αποκλειστικά από το πρώτο στο τελευταίο, με βήμα 1 (ενώ στο απλό `for` όλα αυτά μπορούν να τροποποιηθούν).

Στην περίπτωση δισδιάστατου πίνακα, αντίστοιχα:

```
double [] [] arr2 = {{1.0, 2.0}, {3.0, 4.0}, {5.0, 6.0}};

for (double [] r : arr2)
      for (double el : r)
            System.out.println(el);
```

Στο πρώτο `for`, το `r` αναφέρεται σε κάθε γραμμή του πίνακα, ενώ στο δεύτερο `for`, το `el` αναφέρεται σε κάθε στοιχείο της κάθε γραμμής, άρα σε κάθε στοιχείο του δισδιάστατου πίνακα. *(Υπενθυμίζεται ότι στη Java οι δισδιάστατοι πίνακες είναι ουσιαστικά μονοδιάστατοι πίνακες με στοιχεία άλλους μονοδιάστατους πίνακες).*

Παράδειγμα: Να γραφεί μέθοδος που να δέχεται ακέραιο n, να δημιουργεί δισδιάστατο πίνακα (n x n) ακέραιων τιμών και να αρχικοποιεί το άνω τριγωνικό του μέρος με την τιμή i*j για κάθε στοιχείο (i,j).

```
public int [][] myMethod(int n)
{
   int [][] arr = new int [n][n];   // δε χρειάζεται αρχικοποίηση στο 0

   // μετατροπή του άνω τριγωνικού μέρους μόνο:
   for (int i=0; i<n; i++)
      for (int j=i; j<n; j++)
         arr[i][j] = i*j;

   return arr;
}
```

Σε περίπτωση που η παραπάνω μέθοδος ανήκε σε μια κλάση `SuppClass` και θέλαμε να την καλέσουμε από μια άλλη κλάση `AppClass`, θα είχαμε κάτι σαν το παρακάτω:

```
public class AppClass
{
   public static void main(String [] args)
   {
      // δήλωση ενός δισδιάστατου πίνακα ακεραίων (βλέπε 1η παρατήρηση παρακάτω)
      int [][] myArray;

         // δημιουργία αντικειμένου της κλάσης SuppClass
      SuppClass obj = new SuppClass();

         // κλήση της μεθόδου για μορφοποίηση του άνω τριγωνικού μέρους του πίνακα
      myArray = obj.myMethod(10);

         // στο σημείο αυτό, ο πίνακας myArray έχει πάρει τιμές
         // όπως προβλέπει η μέθοδος myMethod της κλάσης SuppClass
   }
}
```

Παρατηρήσεις:

- Η εντολή δήλωσης και δημιουργίας ενός πίνακα που αναφέρθηκε στην αρχή της ενότητας, ουσιαστικά είναι δύο εντολές: η *δήλωση* του πίνακα (π.χ., `int [][] myArray;`) και η *δημιουργία* του με τη `new` (π.χ. `myArray = new int[10][10];`). Στο συγκεκριμένο παράδειγμα είναι περιττό να δημιουργηθεί ο πίνακας `myArray`, αφού στην ουσία δημιουργείται και παίρνει τιμές μέσα στη μέθοδο `myMethod`, η οποία τον επιστρέφει. Πρέπει όμως μέσα στη `main` να δηλωθεί ο συγκεκριμένος πίνακας, ώστε στη συνέχεια να αποθηκευτεί σε αυτόν ο πίνακας που επιστρέφεται από τη μέθοδο `myMethod`.
- Αν στην κλάση `AppClass` ο πίνακας `myArray` δηλωνόταν *εκτός* της μεθόδου `main` (δηλαδή ήταν πίνακας κλάσης και όχι τοπικός πίνακας της `main`), τότε θα έπρεπε να ορισθεί ως `static`, γιατί η `main` είναι `static` μέθοδος και `static` μέθοδοι έχουν πρόσβαση μόνο σε `static` στοιχεία (μεταβλητές και πίνακες).

Αντιγραφή πινάκων

```
// δημιουργία δύο πινάκων
int [ ] array1 = new int [5];
int [ ] array2 = new int [5];

// αρχικοποίηση του πίνακα array1
for (int i=0; i<array1.length; i++)
{
   array1[i] = 2*i;   // array1 = {0,2,4,6,8}
}

// προσπάθεια αντιγραφής του πίνακα array1 στον πίνακα array2
array2 = array1;   // Ίδιος χώρος στη μνήμη. Ένας πίνακας με δύο ονόματα!

// Στην ουσία αυτό δεν έκανε αντιγραφή! Αν:
array2[0]=100;
System.out.println(array1[0]); // τυπώνει 100 και όχι 0
// δηλ., αλλάζοντας τον array2 αλλάζει και ο array1.
// Λογικό, αφού πρόκειται για ΕΝΑΝ πίνακα με δύο ονόματα...

// Ο σωστός τρόπος αντιγραφής του array1 στον array2:

System.arraycopy(array1, 0, array2, 0, array1.length);
```

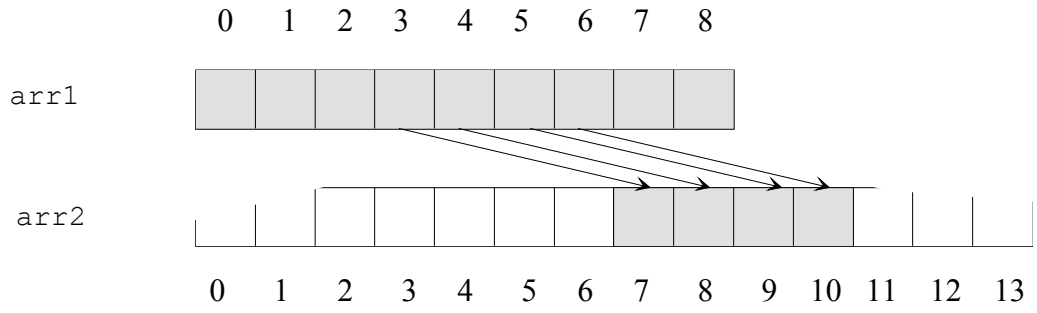
Δηλαδή, για αντιγραφή πινάκων χρησιμοποιούμε τη `System.arraycopy`:

```
System.arraycopy(sourceArray, sourceArrayStartingPosition,
       destinationArray, destinationArrayStartingPosition,
       howManyElementsToCopy);
```

όπου:
`sourceArray`: ο αρχικός πίνακας
`sourceArrayStartingPosition`: από ποιο σημείο να αρχίσει να παίρνει στοιχεία για αντιγραφή
`destinationArray`: ο τελικός πίνακας
`destinationArrayStartingPosition`: από ποιο σημείο να αρχίσει να αντιγράφει στο νέο πίνακα
`howManyElementsToCopy`: πόσα στοιχεία να αντιγραφούν

Άρα μπορούμε να αντιγράψουμε και τμήματα πινάκων, π.χ.:

```
System.arraycopy(arr1, 3, arr2, 7, 4);
```



Πίνακες αντικειμένων

Οι πίνακες, εκτός από στοιχεία πρωτογενών τύπων (`int`, `double`, `char` κτλ.), μπορούν να αποτελούνται και από **στοιχεία αντικείμενα**. Όπως μεταβλητές του ίδιου τύπου "ομαδοποιούνται" σε ένα πίνακα υπό ένα κοινό όνομα, έτσι και αντικείμενα μιας κλάσης μπορούν να ομαδοποιηθούν υπό ένα κοινό όνομα σε ένα πίνακα αντικειμένων. Η μόνη διαφορά με τους κοινούς πίνακες είναι ότι οι πίνακες αντικειμένων είναι τύπου κάποιας κλάσης (και όχι πρωτογενούς τύπου) και τα στοιχεία τους είναι αντικείμενα και όχι απλές μεταβλητές.

Αν έχουμε μια κλάση υποστήριξης `SuppClass`, υπάρχουν δύο βήματα για τη δημιουργία αντικειμένων της κλάσης αυτής που θα αποτελούν στοιχεία πίνακα:

i) δημιουργία πίνακα:

```
SuppClass [ ] objects = new SuppClass [5];
```

Έτσι δημιουργείται ο πίνακας `objects` με μέγεθος 5, του οποίου τα στοιχεία είναι τύπου `SuppClass`, δηλαδή αντικείμενα της κλάσης `SuppClass`. Άρα, ο πίνακας `objects` θα περιέχει 5 αντικείμενα της κλάσης `SuppClass`.

ii) δημιουργία αντικειμένων:

```
for (int i=0; i<objects.length; i++)
   objects[i] = new SuppClass();
```

ή

```
for (SuppClass obj : objects)
   obj = new SuppClass();
```

Έτσι, δημιουργήθηκαν τα 5 αντικείμενα.

Οπότε, για κλήση π.χ. της μεθόδου `method1()` της `SuppClass` μέσω του αντικειμένου `objects[2]`:

```
objects[2].method1();
```

ή της `method2()` από κάποιο άλλο αντικείμενο, π.χ. το `objects[0]`:

```
objects[0].method2();
```

Μέθοδοι πρόσβασης που επιστρέφουν πίνακες:

```
...
// δημιουργία ενός private πίνακα:
private double [][] a = new double [5][10];
...

// μέθοδος πρόσβασης που επιστρέφει τον private πίνακα
public double [][] getA()
{
   return a;
}
```

Μέθοδοι που δέχονται πίνακες ως παράμετρο εισόδου:

```
public int [] initArray(int [] arr)
{
   for (int a : arr)
      a = 1;

   return arr;
}
```

Η συγκεκριμένη μέθοδος δέχεται έναν μονοδιάστατο πίνακα ακέραιων τιμών, αρχικοποιεί τα στοιχεία του στην τιμή 1 και τον επιστρέφει.

Όταν *δεν* ξέρουμε εκ των προτέρων τη διάσταση του πίνακα που θέλουμε να δημιουργήσουμε:

Π.χ., θέλουμε να έχουμε έναν μονοδιάστατο πίνακα ακέραιων στοιχείων στην κλάση μας, αλλά δε γνωρίζουμε εκ των προτέρων το μέγεθός του, δηλαδή πόσες `int` τιμές θα θέλουμε να κρατάει. Σε αυτή την περίπτωση, μπορούμε να διαχωρίσουμε τη σύνθετη εντολή δήλωσης και δημιουργίας του πίνακα (όπως αναφέρθηκε και παραπάνω), έτσι ώστε το πρώτο της μέρος, δηλαδή η δήλωση του πίνακα, να είναι στην κλάση (πριν τις μεθόδους) και άρα ο πίνακας να ανήκει σε όλη την κλάση (πίνακας αντικειμένου), και το δεύτερο μέρος, που είναι και η πραγματική δημιουργία του πίνακα, να είναι μέσα σε μία μέθοδο που να δέχεται ως παράμετρο εισόδου το μέγεθος του πίνακα:

```
public class Example
{
  private int [] myArray;  // δήλωση του πίνακα αντικειμένου

   . . .

 public void setArray(int x)
  {
     myArray = new int [x]; // δημιουργία του πίνακα
  }

   . . .

  // αν μετά έχω μια μέθοδο που π.χ. αρχικοποιεί αυτόν τον πίνακα:
  public void initArray()
  {
     for (int i=0; i<myArray.length; i++)
        myArray[i] = 1;
  }
}
```

Θα πρέπει να προσέξω να καλέσω πρώτα τη `setArray` και μετά την `initArray`, για οποιοδήποτε αντικείμενο αυτής της κλάσης, διαφορετικά θα υπάρξει NullPointerException (ένα σφάλμα-εξαίρεση που μπορούμε να διαχειριστούμε, όπως θα δούμε σε επόμενη ενότητα), γιατί ο πίνακας στην ουσία δεν έχει δημιουργηθεί πριν γίνει κλήση της `setArray`, άρα το `myArray.length` αλλά και η απόδοση τιμής σε στοιχεία του πίνακα κατά την κλήση της `initArray`, θα ήταν προβληματικά.

ΣΗΜΕΙΩΣΗ: Ένα σημείο που πρέπει να επισημανθεί είναι ότι, όπως έχει ήδη αναφερθεί, οι πίνακες στη Java είναι αντικείμενα, άρα όταν μία μέθοδος δέχεται πίνακα, αυτός μεταβιβάζεται μεν σε αυτή (κατά την κλήση της) "με τιμή" (pass-by-value), αλλά επειδή είναι αντικείμενο, οι όποιες μεταβολές γίνουν σε συγκεκριμένα στοιχεία του, παραμένουν (περνάνε) και στον αρχικό πίνακα της μεθόδου που κάλεσε (και μεταβίβασε) τον πίνακα. Επομένως, σε πολλές περιπτώσεις, δεν υπάρχει λόγος να επιστρέφεται ο πίνακας με την εντολή `return`, δηλαδή η μέθοδος που τον μεταβάλει μπορεί να είναι `void`. Σε άλλες περιπτώσεις όμως (π.χ. αν δημιουργηθεί καινούριος πίνακας ή αντίγραφο του εισερχόμενου πίνακα κτλ.), η επιστροφή του πίνακα με την εντολή `return` είναι απαραίτητη, και η Java, ειδικά επειδή δεν διαθέτει τη δυνατότητα περάσματος με αναφορά (pass-by-reference), δίνει αυτή τη δυνατότητα επιστροφής.

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 8

Πακέτα (Packages)

Τα πακέτα αποτελούν τρόπο ομαδοποίησης κλάσεων συναφούς λειτουργικότητας.

Ένα πακέτο λειτουργεί σαν βιβλιοθήκη κλάσεων που μπορούν να χρησιμοποιηθούν από ένα πρόγραμμα χωρίς να βρίσκονται στον υποκατάλογό του.

Κάθε πακέτο έχει ένα όνομα και όλες του οι κλάσεις βρίσκονται στον ίδιο υποκατάλογο του συστήματος αρχείων του υπολογιστή, που έχει το ίδιο όνομα με το πακέτο.

Οι κλάσεις ενός πακέτου είναι κανονικές κλάσεις, άρα ορίζονται:

```
public class <ClassName>
```

αλλά **πριν** τον ορισμό τους έχουν την ακόλουθη δήλωση:

```
package <packageName>;
```

π.χ.
Έστω ότι είμαστε μέσα στον υποκατάλογο με όνομα `pack1`, ο οποίος έχει μέσα δύο κλάσεις, τα αρχεία ClassOne.java και ClassTwo.java:

Αρχείο ClassOne.java:

```
package pack1;

public class ClassOne
{
   ...
   ...
}
```

Αρχείο ClassTwo.java:

```
package pack1;

public class ClassTwo
{
   ...
   ...
}
```

Επίσης, έστω ότι μέσα στον υποκατάλογο `pack1` υπάρχει νέος υποκατάλογος με το όνομα `smallPack` ο οποίος περιέχει τις ακόλουθες κλάσεις:

Αρχείο Class1.java:

```
package pack1.smallPack;

public class Class1
{
  ...
  ...
}
```

Αρχείο Class2.java:

```
package pack1.smallPack;

public class Class2
{
  ...
  ...
}
```

Για να χρησιμοποιήσουμε στο πρόγραμμά μας τις κλάσεις του πακέτου `pack1` χρησιμοποιούμε την εντολή **`import`**:

Αρχείο MyClass1.java:

```
import pack1.*;

public class MyClass1
{
  ...
  ...
}
```

ενώ ειδικά για τις κλάσεις του πακέτου `smallPack`:

Αρχείο MyClass2.java:

```
import pack1.smallPack.*;

public class MyClass2
{
  ...
  ...
}
```

Άρα, με τη δήλωση: `import <packageName>.*;` μπορούμε να χρησιμοποιήσουμε όλες τις κλάσεις ενός πακέτου σε κάποια κλάση του προγράμματός μας. Είναι κλάσεις που βρίσκονται εκτός του υποκαταλόγου του προγράμματός μας.

- Το `.*` δηλώνει όλες τις κλάσεις ενός πακέτου.
- Μπορούμε να κάνουμε `import` σε μια κλάση μόνο συγκεκριμένες κλάσεις ενός πακέτου, ενώ μπορούμε φυσικά να κάνουμε `import` κλάσεις από *πολλά* πακέτα. Π.χ., αν θέλαμε

στην κλάση μας `MyClass3` να συμπεριλάβουμε όλο το `pack1` αλλά και την `Class2` του `pack1.smallPack`:

```
import pack1.*;
import pack1.smallPack.Class2;

public class MyClass3
{
   ...
}
```

Στο παράδειγμα αυτό των `pack1` και `smallPack` πακέτων, το `smallPack`, το οποίο βρίσκεται σε έναν υποκατάλογο του `pack1`, ονομάζεται υποπακέτο του `pack1`. Η δομή αυτή υπάρχει για να μπορούν οι κλάσεις κάθε πακέτου να χωρίζονται σε υπο-ομάδες συναφούς λειτουργικότητας, έτσι ώστε κάποιος να μπορεί εύκολα να συμπεριλάβει στο πρόγραμμά του μια υπο-ομάδα κλάσεων με τη δήλωση:

```
import <πακέτο>.<υποπακέτο>.*;
```

χωρίς να χρειάζεται να κάνει `import` μία-μία τις κλάσεις αυτές ή να αναγκάζεται να κάνει `import` όλο το πακέτο, το οποίο μπορεί να περιέχει πολλές κλάσεις που δεν θα χρειαστεί.

Σημείωση: Η εντολή `import <πακέτο>.*;` κάνει `import` μόνο τις κλάσεις του πακέτου και *όχι* τυχόν υποπακέτα που υπάρχουν μέσα σε αυτό, τα οποία θα πρέπει να γίνουν ξεχωριστά `import` σε περίπτωση που κάποιος θέλει να τα συμπεριλάβει και αυτά.

Περί ορατότητας μεταβλητών και μεθόδων:

Ορατότητα:	`public`	`private`	χωρίς δήλωση ορατότητας
Από την ίδια την κλάση	ναι	ναι	ναι
Από άλλη κλάση στο ίδιο πακέτο (ή απλά στο ίδιο πρόγραμμα)	ναι	όχι	**ναι**
Από άλλη κλάση έξω από το πακέτο (η οποία κάνει import το πακέτο)	ναι	όχι	**όχι**

δηλαδή,

- οι `public` μεταβλητές (ή μέθοδοι) είναι ορατές από παντού
- οι `private` είναι ορατές μόνο μέσα στην κλάση όπου ορίζονται
- οι "χωρίς δήλωση" είναι ορατές από τις κλάσεις του πακέτου μόνο, δηλαδή αποτελούν κάτι ενδιάμεσο, μεταξύ `public` και `private`.

Άρα: - για κλάσεις του ίδιου πακέτου: «χωρίς δήλωση» ≡ `public`
- για κλάσεις εκτός πακέτου: «χωρίς δήλωση» ≡ `private`

(Υπάρχει και η ορατότητα `protected` στην οποία θα αναφερθούμε στην ενότητα της κληρονομικότητας)

Παράδειγμα:

Έχουμε ένα πακέτο (pack1) με τις ακόλουθες δύο κλάσεις:

```
package pack1;
public class ClassA
{
  private void method1()
   { ... }

  public void method2()
   { ... }

  void method3()
   { ... }
}
```

```
package pack1;
public class ClassB
{

   ClassA obj = new ClassA();

   public void method4()
   {
    obj.method1(); // ΛΑΘΟΣ!
    obj.method2();
    obj.method3();
   }
}
```

Στο πρόγραμμά μας κάνουμε χρήση του πακέτου `pack1`:

```
import pack1.*;
public class MyClass
{
  ClassA objA = new ClassA();

  public void myMethod()
   {
    objA.method1(); // ΛΑΘΟΣ!
    objA.method2();
    objA.method3(); // ΛΑΘΟΣ!!
    objA.method4(); // ΛΑΘΟΣ!! (η method4() ανήκει στην ClassB)
   }
}
```

Κλάσεις με το ίδιο όνομα σε διαφορετικά πακέτα

Αν υπάρχει κλάση με το ίδιο όνομα σε δύο διαφορετικά πακέτα, τότε αναφορές στο όνομα της κλάσης αυτής θα πρέπει να περιέχουν και το όνομα του πακέτου. Π.χ., αν, σε συνέχεια του προηγούμενο παραδείγματος με το `pack1`, θέλουμε να χρησιμοποιήσουμε και το `pack2`, το οποίο περιέχει και αυτό μια κλάση με το όνομα π.χ. `ClassB`, τότε:

```
import pack1.*;
import pack2.*;
public class MyNewClass
{
      pack1.ClassB obj1 = new pack1.ClassB();
      pack2.ClassB obj2 = new pack2.ClassB();
      .
      .
      .
}
```

Στην περίπτωση αυτή, χρησιμοποιούμε το `<πακέτο>.<Κλάση>` για να αναφερθούμε στη συγκεκριμένη κάθε φορά ClassB επειδή υπάρχουν δύο τέτοιες κλάσεις. Επιπλέον, επειδή ακριβώς χρησιμοποιούμε το `<πακέτο>.<Κλάση>`, δεν είναι απαραίτητο να κάνουμε `import` τα πακέτα pack1 και pack2.

Δηλαδή, στις κλάσεις που χρησιμοποιούμε την κλάση `Scanner` για είσοδο από το πληκτρολόγιο και κάνουμε `import` το πακέτο `java.util`:

```
import java.util.*;
public class InputExample
{
   Scanner input = new Scanner(System.in);
   .
   .
}
```

θα μπορούσαμε ισοδύναμα να γράφαμε:

```
public class InputExample
{
   java.util.Scanner input = new java.util.Scanner(System.in);
   .
   .
}
```

→ Σε μια κλάση μπορεί να έχουμε και δήλωση πακέτου και `import` κάποιου πακέτου. Π.χ.

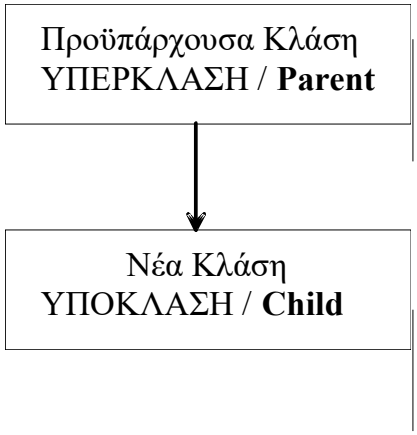
```
package pack3.test;   // η κλάση ανήκει στο πακέτο pack3.test
import javax.swing.*; // η κλάση χρησιμοποιεί το πακέτο javax.swing

public class AskForData
{
  ...
}
```

Η δήλωση πακέτου (`package <packageName>;`) είναι υποχρεωτικά η πρώτη εντολή της κλάσης.

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 9

Κληρονομικότητα (Inheritance)

Υπάρχουν κλάσεις που εμπεριέχουν μεταβλητές και μεθόδους που έχουν οριστεί σε προϋπάρχουσες κλάσεις:



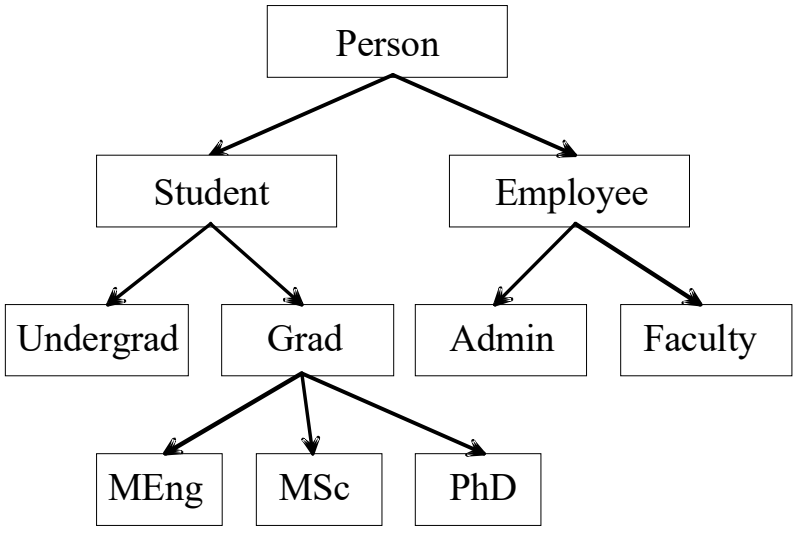

```
public class Child extends Parent
{
...
}
```

Η δήλωση λοιπόν της σχέσης κληρονομικότητας μεταξύ δύο κλάσεων είναι της μορφής:

```
public class <Υποκλάση> extends <Υπερκλάση>
```

Έτσι δημιουργείται μία ιεραρχία, όπου από τη γενικότερη κλάση μπορούμε να περάσουμε σε ειδικότερες, όπου κάθε μία έχει και τα χαρακτηριστικά της γενικότερης προγόνου της, αλλά και αυτά των ακόμα γενικότερων προγόνων της προγόνου της.

Π.χ., θέλουμε να φτιάξουμε ένα πρόγραμμα που να περιέχει στοιχεία για τα μέλη ενός πανεπιστημίου. Μια ομαδοποίηση των μελών μπορεί να είναι η εξής:



Στο παράδειγμα αυτό, θα είχαμε δηλώσεις κλάσεων της μορφής:

```
public class Person
{
  ...
}

public class Student extends Person
{
  ...
}

public class Undergrad extends Student
{
  ...
}

public class Grad extends Student
{
  ...
}
```

κτλ.

Και τα αντικείμενα της `Student` *και* τα αντικείμενα της `Employee` αντιπροσωπεύουν ανθρώπους, άρα έχουν κάποιες κοινές ιδιότητες. Επομένως, οι μέθοδοι που θα αρχικοποιούν, θα τροποποιούν ή θα κάνουν output π.χ. το όνομα κάποιου μέλους του πανεπιστημίου, είτε είναι φοιτητής (Student) είτε είναι εργαζόμενος (Employee), θα είναι ίδιες. Όλες αυτές τις μεθόδους τις βάζουμε σε μία υπερκλάση `Person`. Εκεί φυσικά βάζουμε και όλα τα κοινά στοιχεία (μεταβλητές) των δύο «κατηγοριών» (φοιτητών και εργαζόμενων). Με αυτή τη λογική δομούνται οι σχέσεις κληρονομικότητας των κλάσεων.

Μια απλή μορφή της κλάσης `Person` είναι η εξής:

```
public class Person
{
        // class variables
  // μεταβλητή για το όνομα του κάθε ατόμου:
  private String name;

        // constructors
  public Person()
  {
    name = "Δεν έχει ορισθεί ακόμα όνομα.";
  }

  public Person(String initName)
  {
    name = initName;
  }

  // setter μέθοδος για αλλαγή του ονόματος
  public void setName(String newName)
  {
    name = newName;
  }
```

```
  // getter μέθοδος
  public String getName()
  {
    return name;
  }

  // μέθοδος για την εκτύπωση του αποτελέσματος
  public void writeOutput()
  {
    System.out.println("Όνομα: " + name);
  }

}
```

Η κλάση `Student` κληρονομεί την κλάση `Person` (τις μεταβλητές της (το `name` δηλαδή) και τις μεθόδους της) και προσθέτει μια δικιά της μεταβλητή για τον αριθμό μητρώου του κάθε φοιτητή:

```
public class Student extends Person
{
  // μεταβλητή για τον αριθμό μητρώου του κάθε φοιτητή
  private int am;

  public Student()
  {
    super();        // -> κλήση του constructor της Person
    am = 0;
  }

  public Student(String initName, int initAM)
  {
    super(initName);  // -> κλήση του constructor της Person
    am = initAM;
  }

  public void reset(String newName, int newAM)
  {
    setName(newName);  // -> κλήση μεθόδου της Person!
    am = newAM;
  }

  public void setAM(int newAM)
  {
    am = newAM;
  }

  public int getAM()
  {
    return am;
  }

  // μέθοδος για την εκτύπωση του αποτελέσματος
  public void writeOutput()
  {
    System.out.println("Όνομα: " + getName());
    System.out.println("Αριθμός Μητρώου: " + am);
  }

}
```

Αφού η `Student` κληρονομεί την `Person`, στην ουσία υπάρχουν δύο μέθοδοι `writeoutput()` μέσα στη `Student`. Αν σε μια υποκλάση υπάρχει μέθοδος με ίδιο όνομα και ίδιο πλήθος και είδος παραμέτρων με κάποια μέθοδο της υπερκλάσης της, τότε η μέθοδος της υποκλάσης κάνει override (υπερκαλύπτει ή παρακάμπτει) τη μέθοδο της υπερκλάσης. Το φαινόμενο αυτό λέγεται **overriding (υπερκάλυψη, ή παράκαμψη)**. Αν μέσα στη `Student` γίνει κάποια κλήση της `writeOutput`, αυτή που τελικά καλείται είναι η `writeOutput()` της `Student` και όχι της `Person`.

Αν η μέθοδος που παρακάμπτεται (της υπερκλάσης) επιστρέφει (`return`) κάποια μεταβλητή, η νέα μέθοδος που την κάνει override (στην υποκλάση) πρέπει να επιστρέφει μεταβλητή του ίδιου τύπου.

→ **Overriding (υπερκάλυψη)** / **Overloading (υπερφόρτωση)**

- Ίδιο όνομα μεθόδων	- Ίδιο όνομα μεθόδων
- και ίδιο πλήθος παραμέτρων	- και διαφορετικό πλήθος παραμ.
- και ίδιο είδος παραμέτρων	- ή διαφορετικό είδος παραμ.
- και ίδιο είδος επιστροφής (return)	- ανεξαρτήτως είδους επιστροφής

Overriding: μεταξύ μεθόδων διαφορετικών κλάσεων που συνδέονται με κληρονομικότητα

Overloading: μεταξύ μεθόδων της ίδιας κλάσης ή μεθόδων κλάσεων που συνδέονται με κληρονομικότητα

Παράδειγμα υλοποίησης της κλάσης εφαρμογής του προγράμματος:

Αν είχαμε τις δύο αυτές κλάσεις υποστήριξης (`Person` και `Student`), στην κλάση εφαρμογής θα μπορούσαμε να είχαμε τα εξής:

```
public class InheritanceExample
{
  public static void main (String [ ] args)
  {
    Student s = new Student();
    s.setName("Xxxxx Yyyyyyy"); // -> setName της Person
    s.setAM(2010001); // -> setAM της Student
    s.writeOutput();   // -> writeOutput της Student (Overriding)
  }

}
```

- Η λέξη `final` δηλώνει μέθοδο που δεν μπορεί να γίνει overridden (να υπερκαλυφθεί).

π.χ.
```
public final void specialMethod()
{
    ...
}
```

- Η λέξη `super`:

i) `super.writeOutput(); //` → κλήση μεθόδου που έχει γίνει overridden
(δηλαδή, καλεί τη μέθοδο της υπερκλάσης)

ii) Κλήση του κατασκευαστή της υπερκλάσης: `super();`

ΠΡΟΣΟΧΗ! Η λέξη `super` όταν χρησιμοποιείται για την κλήση ενός κατασκευαστή της υπερκλάσης μέσα σε κάποιον κατασκευαστή της υποκλάσης, πρέπει να είναι η πρώτη εντολή αυτού του κατασκευαστή. Δεν μπορεί να χρησιμοποιηθεί αργότερα μέσα στον κατασκευαστή.
Στην πραγματικότητα, αν δεν υπάρχει κλήση του κατασκευαστή της υπερκλάσης (κάποια μορφή της `super()` δηλαδή) μέσα στον κατασκευαστή της υποκλάσης, τότε η Java καλεί *αυτόματα* τον default constructor της υπερκλάσης.

Άρα, στον κατασκευαστή της `Student`:

```
public Student()
{
  super();
  am = 0;
}
```

θα ήταν το ίδιο αν γράφαμε:

```
public Student()
{
  am = 0;
}
```

Συμβουλή: Καλύτερα να γράφουμε την κλήση της `super()` κι ας μην είναι απαραίτητο, για να είναι πιο σαφής ο κώδικάς μας.

- Η λέξη `this` αναφέρεται στον κατασκευαστή της υποκλάσης (και γενικότερα, στον κατασκευαστή της ίδιας της κλάσης στην οποία βρίσκεται).

Π.χ., ένας ακόμη κατασκευαστής της `Student` θα μπορούσε να είναι ο εξής:

```
public Student(String initName)
{
  this(initName, 0); → κλήση του constructor: Student(String, int)
}
```

Σημείωση: Πρέπει και η κλήση της `this` να είναι η *πρώτη* εντολή του κατασκευαστή. Προφανώς, όταν σε έναν κατασκευαστή της υποκλάσης υπάρχει κλήση άλλου κατασκευαστή της μέσω της `this`, τότε δεν μπορεί να υπάρχει ταυτόχρονα και η εντολή `super`, αφού ο κατασκευαστής της υπερκλάσης θα κληθεί τελικά μέσω του κατασκευαστή στον οποίο αναφέρεται η `this`.

Από αυτά που είπαμε μέχρι τώρα για τη `super`, προκύπτει ότι θα μπορούσαμε να γράψουμε πιο σωστά τη μέθοδο `writeOutput()` της `Student`, χωρίς επανάληψη κώδικα.

Δηλαδή, η αρχική μέθοδος:

```
public void writeOutput()
{
  System.out.println("Ονομα: " + getName());
  System.out.println("Αριθμός Μητρώου: " + am);
}
```

μπορεί να γίνει:

```
public void writeOutput()
{
  super.writeOutput();  // -> κλήση της writeOutput της Person
  System.out.println("Αριθμός Μητρώου: " + am);
}
```

Η ορατότητα `protected`:

Μέχρι τώρα είχαμε δει τις εξής δηλώσεις ορατότητας μεταβλητών ή μεθόδων:

`public`, `private` και χωρίς δήλωση.

Υπάρχει και η δήλωση ορατότητας **`protected`** η οποία κάνει τις μεταβλητές ή μεθόδους ορατές μόνο από κλάσεις που σχετίζονται με κληρονομικότητα (και από κλάσεις του ίδιου πακέτου, όπως θα δούμε). Δηλαδή μια μεταβλητή ή μέθοδος που δηλώνεται ως `protected`, λειτουργεί σαν `public` για τις κλάσεις απογόνους της κλάσης ορισμού της και σαν `private` για όλες τις υπόλοιπες κλάσεις του προγράμματος.

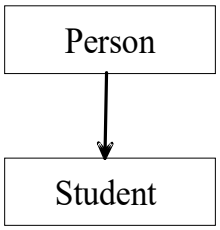
Συνοπτικά:

Ορατότητα:	`public`	`protected`	χωρίς δήλωση ορατότητας	`private`
Μεταξύ μεθόδων της ίδιας κλάσης	ναι	ναι	ναι	ναι
Μεταξύ κλάσεων του ίδιου πακέτου	ναι	**ναι**	ναι	όχι
Μεταξύ κλάσεων εκτός πακέτου	ναι	όχι	όχι	όχι
Από υποκλάσεις του ίδιου πακέτου	ναι	**ναι**	ναι	όχι
Από υποκλάσεις εκτός πακέτου	ναι	**ναι**	όχι	όχι

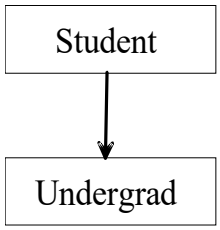
Άρα, γενικά, ως προς το επίπεδο ορατότητας:

`public` > `protected` > χωρίς δήλωση ορατότητας > `private`

Όπως είχαμε τη σχέση κληρονομικότητας:



έτσι μπορούμε να έχουμε και τη σχέση:



Στη δεύτερη αυτή σχέση κληρονομικότητας, η κλάση `Student` γίνεται πλέον <u>υπερκλάση</u> και το ρόλο της υποκλάσης παίζει η κλάση `Undergrad`. Έτσι δημιουργείται μια αλυσίδα κληρονομικότητας που αποτελείται από απλά ζευγάρια κληρονομικότητας, η οποία οδηγεί στην *κληρονομικότητα πολλών επιπέδων*. Η αλυσίδα αυτή ξεκινάει από το «γενικό» και προχωράει προς το «ειδικό». Η «εξειδίκευση» όμως αυτή κάνει τις κλάσεις «ευρύτερες», αφού κληρονομούν τα στοιχεία των γενικότερων κλάσεων και μπορούν να τα χρησιμοποιούν σαν δικά τους.

Ο παρακάτω πίνακας δείχνει την αύξηση του εύρους των διαθέσιμων στοιχείων των κλάσεων, καθώς αυτές εξειδικεύονται και συγχρόνως κληρονομούν στοιχεία των προγόνων τους, για το παράδειγμά μας:

Οι κλάσεις →	`Person`	`Student`	`Undergrad`
χρησιμοποιούν μετ/τές & μεθόδους της κλάσης:			
`Person`	ναι	ναι	**ναι**
`Student`	όχι	ναι	ναι
`Undergrad`	όχι	όχι	ναι

Προφανώς κάθε κλάση χρησιμοποιεί μεταβλητές και μεθόδους του εαυτού της, αλλά βλέπουμε ότι η `Person` χρησιμοποιεί απλά τα δικά της στοιχεία, ενώ η `Student` χρησιμοποιεί τα δικά της και της `Person`, ενώ η `Undergrad` χρησιμοποιεί και τα δικά της και της `Student`, *αλλά και της* `Person`. Όλα αυτά βέβαια αφορούν μεταβλητές και μεθόδους ορατότητας `public` ή `protected`. (όμως κληρονομούνται *και* τα `private` πεδία).

Η κλάση `Undergrad` θα μπορούσε να περιέχει τα ακόλουθα:

```
public class Undergrad extends Student
{

  // μεταβλητή για το έτος φοίτησης του κάθε φοιτητή
  private int year;

  // constructors

  public Undergrad()
  {
    super();  // καλεί τον default constructor της Student
    year = 1;
  }

  public Undergrad(String initName, int initAM, int initYear)
  {
    super(initName, initAM);  // → constructor της Student
    year = initYear;
  }
```

```
  // setter method
  public void reset(String newName, int newAM, int newYear)
  {
    reset(newName, newAM); //→ κλήση της overloaded reset της Student
    year = newYear;
  }

  // setter method
  public void setYear(int newYear)
  {
    year = newYear;
  }

  // getter method
  public int getYear()
  {
    return year;
  }

  // μέθοδος για εκτύπωση του αποτελέσματος
  public void writeOutput()
  {
    super.writeOutput(); // → Η writeOutput της Student – Overriding
    System.out.println("Έτος φοίτησης: " + year);
  }
}
```

Στην κλάση αυτή παρουσιάζονται τόσο το φαινόμενο του overriding όσο και αυτό του overloading. Overriding μπορεί να γίνει και μεταξύ κλάσεων όχι "άμεσης" κληρονομικότητας αλλά "έμμεσης", δηλαδή μεταξύ της `Undergrad` και της `Person` στο παράδειγμά μας (οι κλάσεις αυτές έχουν *έμμεση σχέση κληρονομικότητας* γιατί η `Undergrad` κληρονομεί τη `Student` και η `Person` κληρονομείται από τη `Student`). Π.χ., θα μπορούσε στην `Undergrad` να υπάρχει και η ακόλουθη μέθοδος:

```
public void setName(String newName)   // μέθοδος της Undergrad
{
   System.out.println("Αλλαγή ονόματος προπτυχιακού φοιτητή");
   super.setName(newName);    // -> setName της Person!
}
```

Η μέθοδος αυτή της `Undergrad` *υπερκαλύπτει τη μέθοδο* `setName` ***της Person*** (παρόλο που η `Person` σε σχέση με την `Undergrad` είναι δύο "γενιές" πίσω). Προσοχή χρειάζεται στο ότι και εδώ η κλήση υπερκαλυμμένης μεθόδου γίνεται με τη χρήση της `super`, παρόλο που η μέθοδος αυτή δεν βρίσκεται στην υπερκλάση (`Student`) αλλά έχει κληρονομηθεί εκεί από μια δική της υπερκλάση (`Person`). Ουσιαστικά δηλαδή, η `super` μπορεί να αναφέρεται και σε μεθόδους *που έχει κληρονομήσει* η υπερκλάση.

(Φυσικά η ύπαρξη της συγκεκριμένης αυτής μεθόδου `setName` *στην κλάση* `Undergrad` *δεν έχει νόημα, αφού στην ουσία δεν κάνει κάτι διαφορετικό από τη μέθοδο* `setName` *της* `Person`*, η οποία μπορεί φυσικά να χρησιμοποιηθεί με οποιοδήποτε αντικείμενο της* `Undergrad`*, αφού κληρονομείται από αυτή (και είναι* `public`*)).*

Προχωρημένα θέματα κληρονομικότητας:

Όπως προαναφέρθηκε, η κληρονομικότητα *ορίζεται* μεταξύ δύο και μόνο κλάσεων,

από υπερκλάση → σε υποκλάση
ή αλλιώς, από parent → σε child

και ο ορισμός γίνεται στην υποκλάση (child) ως εξής:

```
public class Child extends Parent
```

αλλά ουσιαστικά *ισχύει σε πολλαπλά επίπεδα*, όταν μια κλάση που κληρονομεί κάποια άλλη κλάση, κληρονομείται με τη σειρά της από κάποια τρίτη κλάση, κοκ. Δηλαδή, στη Java δεν υπάρχει *πολλαπλή κληρονομικότητα* όπως υπάρχει σε άλλες γλώσσες προγραμματισμού (π.χ. C++), αλλά υπάρχει *κληρονομικότητα πολλών επιπέδων*. Από τα όσα έχουν αναφερθεί μέχρι τώρα, γίνεται σαφές ότι ο βασικός σκοπός της κληρονομικότητας είναι η *επαναχρησιμοποίηση κώδικα*.

Μετατροπή τύπου αντικειμένων

Στη Java τα αντικείμενα της κλάσης `Parent` είναι μόνο τύπου `Parent`, ενώ τα αντικείμενα της κλάσης `Child` είναι τύπου `Child` *και* τύπου `Parent`. Στην ουσία, εάν μία κλάση έχει πολλούς προγόνους (έμμεσα, μέσω της κλάσης που κληρονομεί η υπερκλάση της, κ.ο.κ.), τα αντικείμενά της είναι τύπου της ίδιας της κλάσης, τύπου της υπερκλάσης της, αλλά και τύπου κάθε μιας από τις κλάσεις που είναι πρόγονοί της. Άρα, στο προηγούμενο παράδειγμα, τα:

```
Person sp1 = new Student();
```

και

```
Student us1 = new Undergrad();
```

είναι **σωστά** γιατί ένα αντικείμενο `sp1` της `Student` μπορεί να είναι και τύπου `Person`, όπως και ένα αντικείμενο `us1` της `Undergrad` μπορεί να είναι και τύπου `Student`, ενώ το ίδιο ισχύει και για το:

```
Person up1 = new Undergrad();
```

αφού τα αντικείμενα της `Undergrad` μπορούν να είναι *και* τύπου `Student` αλλά *και* τύπου `Person` (αφού και οι δύο αυτές κλάσεις είναι πρόγονοί της).

Αντιθέτως, το:

```
Student ps1 = new Person();  // ΛΑΘΟΣ!
```

είναι **λάθος**, αφού τα αντικείμενα μιας κλάσης `Parent` (εδώ το `ps1` της `Person`) είναι μόνο τύπου της κλάσης αυτής και όχι τύπου των απογόνων της.

Τι νόημα όμως έχει μια δημιουργία αντικειμένου τύπου `Parent` με τον κατασκευαστή μιας `Child`, όπως π.χ. η εντολή:

```
Person sp1 = new Student();
```

Μια πιο σωστή διατύπωση της έκφρασης "*δημιουργία αντικειμένου τύπου `Person` με τον κατασκευαστή της `Student`"* θα ήταν η: "*δημιουργία αντικειμένου της `Student` που δηλώνεται ως τύπου `Person`"*. Στην ουσία, αυτό που γίνεται είναι μια μετατροπή (*type casting*) ενός αντικειμένου της `Student` σε τύπο `Person`. Το αντικείμενο αυτό (`sp1`) είναι πρακτικά κάτι ενδιάμεσο από αντικείμενο της `Person` και της `Student`. Μπορεί να χρησιμοποιηθεί ως όρισμα εισόδου σε μεθόδους που δέχονται αντικείμενα τύπου `Person` αλλά όχι σε μεθόδους που δέχονται αντικείμενα τύπου `Student`. Αντιθέτως, ένα κανονικό αντικείμενο της `Student` θα μπορούσε να χρησιμοποιηθεί και στις δύο περιπτώσεις. Μια άλλη διαφορά θα γίνει σαφής στην αμέσως επόμενη ενότητα.

Το συγκεκριμένο casting ονομάζεται *upcasting*, επειδή μία κλάση τύπου `Child` "αναβαθμίζεται" σε τύπου `Parent`. Το αντίθετο (*downcasting*) είναι δυνατό με την εντολή:

```
Student s1 = (Student) sp1;
```

Έτσι μετατρέπεται το αντικείμενο `sp1` σε ένα κανονικό αντικείμενο της `Student`.

Παρατήρηση: Εάν η μετατροπή εφαρμοζόταν σε ένα κανονικό αντικείμενο της `Person` (π.χ. στο `Person p = new Person()`), δηλαδή:

```
Student s2 = (Student) p;
```

τότε η εντολή θα γινόταν compile αλλά θα προέκυπτε *σφάλμα εκτέλεσης* (run-time error). Δηλαδή, downcasting γίνεται μόνο σε αντικείμενα που έχουν υποστεί upcasting (ή έχουν δημιουργηθεί με αυτόν τον τρόπο, όπως τα `sp1`, `up1` κλπ παραπάνω).

Παράδειγμα:

Έστω ότι υπάρχει στην κλάση `Student` η ακόλουθη μέθοδος `equals` που συγκρίνει δύο φοιτητές και επιστρέφει `true` αν έχουν τους ίδιους αριθμούς μητρώου:

```
public boolean equals(Student otherStudent)
{
  return (this.am == otherStudent.am);
}
```

ενώ στην κλάση `Undergrad` υπάρχει η ακόλουθη έκδοσή της που την κάνει overload (υπερφόρτωση), αφού έχει διαφορετικού τύπου παράμετρο εισόδου:

```
public boolean equals(Undergrad otherUndergrad)
{
  return (super.equals(otherUndegrad) && (this.year == otherUndergrad.year) );
}
```

Παρατηρούμε ότι η `equals` της `Undergrad` (η οποία επιστρέφει `true` εάν δύο φοιτητές έχουν τον ίδιο αριθμό μητρώου και είναι στο ίδιο έτος) καλεί την `equals` της `Student` χρησιμοποιώντας τη λέξη `super`, παρόλο που δεν πρόκειται για μια μέθοδο που έχει υπερκαλυφθεί (override) αλλά για μια μέθοδο που έχει υπερφορτωθεί (overload)! Η λέξη `super` είναι απαραίτητη διότι ο σκοπός είναι να κληθεί η `equals` της `Student` με ένα αντικείμενο `Undergrad` ώστε να ελεγχθεί η ταύτιση των αριθμών μητρώου. Αυτό μπορεί να γίνει (παρόλο που η `equals` της `Student` δέχεται

αντικείμενο τύπου `Student`) γιατί ένα αντικείμενο τύπου `Undergrad` είναι *και* τύπου `Student`. Μια κλήση όμως της `equals` με το `otherUndergrad` χωρίς τη `super` θα έδινε προτεραιότητα στην `equals` της `Undergrad` που δέχεται αντικείμενο τύπου `Undergrad` (δηλαδή η μέθοδος θα καλούσε τον εαυτό της). Στην ουσία ένα αντικείμενο τύπου `Undergrad` είναι πρώτα τύπου `Undergrad` και μετά τύπου `Student`.

Αλλαγή ορατότητας υπερκαλυμμένης (overriden) μεθόδου

Μια μέθοδος που υπερκαλύπτεται σε κάποια υποκλάση, μπορεί να αλλάξει ορατότητα, δηλαδή μπορεί ο προγραμματιστής να αλλάξει τα δικαιώματα προσπέλασης μιας υπερκαλυμμένης μεθόδου. Η αλλαγή μπορεί να γίνει μόνο προς την κατεύθυνση της διεύρυνσης της ορατότητας της μεθόδου, π.χ. από `private` σε `public` ή από `protected` σε `public` και ποτέ προς την αντίθετη κατεύθυνση. Άρα, μια μέθοδος της υπερκλάσης:

```
private void doSomething()
```

θα μπορούσε να υπερκαλυφθεί στην υποκλάση από την ακόλουθη μέθοδο:

```
public void doSomething()
```

Παρατήρηση: Στο παράδειγμα δημιουργίας ενός αντικειμένου της `Student` τύπου `Person` που αναφέρθηκε παραπάνω (`Person sp1 = new Student();`), μια κλήση της μεθόδου `writeOutput` με αυτό το αντικείμενο (π.χ. από την κλάση εφαρμογής), θα αναφερόταν όπως προαναφέρθηκε στη `writeOutput` της `Student`, αφού αυτή της `Person` έχει υπερκαλυφθεί (όπως θα γινόταν και με ένα κανονικό αντικείμενο της `Student` δηλαδή). Όμως, εάν η `writeOutput` της `Person` είχε ορισθεί ως `private` (ενώ αυτή της `Student` κανονικά ως `public`), τότε δεν θα ήταν δυνατή κλήση της `writeOutput` της `Student` με αυτό το αντικείμενο, παρόλο που η `writeOutput` της `Student` η οποία καλείται, είναι `public`. *Άρα, στην περίπτωση κλήσης μεθόδου μιας υποκλάσης που υπερκαλύπτει κάποια μέθοδο υπερκλάσης, με χρήση αντικειμένου της υποκλάσης "τύπου της υπερκλάσης" (εδώ, της `Student` τύπου `Person`), η δυνατότητα κλήσης της μεθόδου αυτής καθορίζεται από την ορατότητα της υπερκαλυμμένης μεθόδου στην υπερκλάση (εδώ, στην `Person`).*

Αλλαγή τύπου υπερκαλυμμένης (overriden) μεθόδου

Από την έκδοση 5.0 της Java και μετά, εκτός από αλλαγή ορατότητας, μια υπερκαλυμμένη μέθοδος μπορεί να υποστεί και αλλαγή τύπου (επιστρεφόμενης τιμής). Σύμφωνα με τον γενικό κανόνα της υπερκάλυψης (override), η μέθοδος της υποκλάσης που υπερκαλύπτει τη μέθοδο της υπερκλάσης, πρέπει να είναι του ίδιου τύπου με τη μέθοδο που υπερκαλύπτεται. Στον κανόνα αυτό υπάρχει η εξής εξαίρεση: *εάν η μέθοδος που υπερκαλύπτεται είναι τύπου κλάσης (δηλαδή, επιστρέφει αντικείμενο), τότε η μέθοδος που την υπερκαλύπτει μπορεί να είναι τύπου οποιασδήποτε κλάσης απογόνου τής αρχικά επιστρεφόμενης κλάσης.* Για παράδειγμα, εάν σε μια κλάση `Parent` υπάρχει η μέθοδος `getUnivMember` που επιστρέφει αντικείμενο τύπου `Student` (κάποιον φοιτητή):

```
public class Parent
{     ...
   public Student getUnivMember(int id)
   {
      ...
   }
}
```

σε μια υποκλάση της `Parent` θα μπορούσε να υπάρχει μια `getUnivMember` που την υπερκαλύπτει, ως εξής:

```
public class Child extends Parent
{
     ...
   public Undergrad getUnivMember(int id)
   {
      ...
   }
     ...
}
```

Η μέθοδος που υπερκαλύπτει την αρχική έκδοση της μεθόδου `getUnivMember` επιστρέφει αντικείμενο τύπου `Undergrad` (δηλαδή κάποιον προπτυχιακό φοιτητή). Αυτό επιτρέπεται γιατί η `Undergrad` κληρονομεί τη `Student`.

Στην πραγματικότητα, αυτό που συμβαίνει στη μέθοδο που υπερκαλύπτει την αρχική, δεν είναι μια απλή αλλαγή τύπου, αλλά η προσθήκη επιπλέον περιορισμών στον επιστρεφόμενο τύπο της. Κάθε αντικείμενο της `Undergrad` είναι στην ουσία ένα αντικείμενο της `Student` με κάποιες επιπρόσθετες ιδιότητες, οι οποίες ενώ είναι ιδιότητες κάθε προπτυχιακού φοιτητή, δεν είναι ιδιότητες που έχει ο κάθε φοιτητής γενικά.

Βασικός λόγος που τα `private` πεδία δεν είναι ορατά στους απογόνους

Όπως ήδη αναφέρθηκε, τα `private` στοιχεία μιας υπερκλάσης, κληρονομούνται μεν στις υποκλάσεις της, αλλά δεν είναι ορατά από αυτές. Π.χ., στο αρχικό παράδειγμα με τις κλάσεις `Person` και `Student`, η μεταβλητή `name` της `Person` είναι `private`. Κληρονομείται φυσικά στη `Student`, δηλαδή κάθε αντικείμενο της `Student` έχει μια μεταβλητή `name` (ο κάθε φοιτητής έχει όνομα), όμως η `Student` για να την προσπελάσει χρησιμοποιεί τις ανάλογες `public` μεθόδους της `Person` (τη setter `setName` και τη getter `getName`). Ο βασικός λόγος που συμβαίνει αυτό είναι γιατί σε διαφορετική περίπτωση, *κάποιος που θα ήθελε να αποκτήσει άμεση πρόσβαση σε οποιαδήποτε `private` μεταβλητή/μέθοδο μιας κλάσης, θα μπορούσε να το κάνει απλά φτιάχνοντας μια υποκλάση της!* Θα καταστρατηγούνταν δηλαδή η έννοια της *ενθυλάκωσης* (encapsulation), μιας από τις πολύ βασικές αρχές του αντικειμενοστραφούς προγραμματισμού που αναφέρεται στην απόκρυψη των "ιδιωτικών" στοιχείων μιας κλάσης από τις υπόλοιπες κλάσεις του προγράμματος.

Η κλάση `Object` και η μέθοδος `toString`

Στη Java, κάθε κλάση είναι απόγονος της κλάσης `Object`. Έτσι, κάθε αντικείμενο οποιασδήποτε κλάσης είναι και τύπου `Object` (αλλά και τύπου όλων των προγόνων τής κλάσης, εάν η συγκεκριμένη κλάση κληρονομεί κάποια άλλη κλάση). Αυτό επιτρέπει τη δημιουργία μεθόδων με παράμετρο εισόδου τύπου `Object`, οι οποίες μπορούν να δεχτούν αντικείμενα οποιασδήποτε κλάσης.

Μία πολύ χρήσιμη μέθοδος της κλάσης `Object` είναι η `toString`. Η μέθοδος αυτή επιστρέφει το περιεχόμενο των μεταβλητών ενός αντικειμένου (οποιασδήποτε κλάσης, αφού όλες οι κλάσεις την κληρονομούν από την `Object`) σε μορφή `String`. Όμως, παρόλο που η μέθοδος αυτή κληρονομείται σε οποιαδήποτε κλάση, δεν μπορεί να χρησιμοποιηθεί άμεσα, γιατί δεν καταφέρνει να κωδικοποιήσει σωστά τις υπάρχουσες κάθε φορά μεταβλητές ενός αντικειμένου σε `String`. *Θα πρέπει να υπάρχει μέθοδος `toString` που να την υπερκαλύπτει.* Στο παράδειγμα με τις κλάσεις

`Person -> Student -> Undergrad` κτλ., αντί των μεθόδων `writeOutput`, θα ήταν καλύτερα να είχαμε μεθόδους `toString` που να κάνουν ακριβώς τα ίδια πράγματα (δηλ. να περιέχουν τον ίδιο κώδικα με τις `writeOutput`) και να επιστρέφουν το επιθυμητό `String` αντί να το εκτυπώνουν. Ο βασικός λόγος είναι ότι οι μέθοδοι `toString` που υπερκαλύπτουν την `toString` της `Object`, αποκτούν μια πολύ "βολική" ιδιότητα αυτής της `toString`: *Καλούνται **αυτόματα** (όταν υπάρχουν σε μια κλάση φυσικά) όταν κάποιος προσπαθήσει να εμφανίσει τα περιεχόμενα ενός αντικειμένου μιας κλάσης, βάζοντας το αντικείμενο μέσα σε μια `System.out.println`.*

Π.χ., στο αρχικό παράδειγμα, εάν στην κλάση `Student` η `writeOutput` μετατρέπονταν στην αντίστοιχη `toString`:

```
public String toString()
{
  return ("Όνομα: " + getName() + "\nΑριθμός Μητρώου: " + am);
}
```

η εντολή που κανονικά θα έπρεπε να υπάρχει στη `main` για να καλεί αυτή την `toString` και να εμφανίζει το αποτέλεσμα στην οθόνη:

```
System.out.println(s.toString());
```

θα μπορούσε να αντικατασταθεί από την:

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

(υπενθυμίζεται ότι το `s` είναι ένα αντικείμενο της `Student`)

Σημείωση: εάν δεν υπήρχε η έκδοση της `toString` στη `Student` που υπερκαλύπτει την `toString` της `Object`, τότε η εντολή δεν θα ήταν λάθος, αλλά δεν θα εκτύπωνε το επιθυμητό αποτέλεσμα. Όταν εκτυπώνεται με `System.out.println` ένα αντικείμενο και δεν υπάρχει μέθοδος `toString` στην κλάση του, τότε εκτυπώνεται απλά η θέση μνήμης του αντικειμένου (μέσω της κλήσης της `toString` της `Object`).

Ο τελεστής `instanceof` και η μέθοδος `getClass()`

Ο τελεστής `instanceof` συντάσσεται ως εξής:

```
αντικείμενο instanceof Κλάση
```

και επιστρέφει `true` εάν το αντικείμενο είναι τύπου της συγκεκριμένης Κλάσης και `false` εάν δεν είναι. Προσοχή χρειάζεται στο ότι, όπως προαναφέρθηκε, τα αντικείμενα μιας κλάσης που κληρονομεί κάποια άλλη, είναι τύπου της κλάσης αυτής αλλά και τύπου των προγόνων της. Άρα σε κάθε τέτοια περίπτωση αντικειμένου, ο τελεστής `instanceof` θα επιστρέφει `true`.

Αντιθέτως, η μέθοδος `getClass()` της κλάσης `Object` επιστρέφει την κλάση της οποίας αντικείμενο είναι το αντικείμενο με το οποίο καλείται. Στην ουσία δεν επιστρέφει απαραίτητα τον τύπο του αντικειμένου, αλλά την κλάση με τον κατασκευαστή της οποίας δημιουργήθηκε ένα αντικείμενο.

Χαρακτηριστικό είναι το ακόλουθο παράδειγμα. Εάν έχουν δημιουργηθεί τα αντικείμενα:

```
Student s = new Student();
Person ps = new Student();
Person p = new Person();
```

τότε οι ακόλουθες εντολές εκτυπώνουν αυτά που φαίνονται δίπλα σε σχόλια:

```
System.out.println(s instanceof Student);    // -> true
System.out.println(s instanceof Person);     // -> true
System.out.println(ps instanceof Student);   // -> true
System.out.println(ps instanceof Person);    // -> true
System.out.println(p instanceof Person);     // -> true
System.out.println(p instanceof Student);    // -> false
System.out.println(ps.getClass() == s.getClass());  // -> true
System.out.println(ps.getClass() == p.getClass());  // -> false
```

Η λέξη `final`:

α) Σε μεταβλητές: Δεν αλλάζει η τιμή τους μετά την αρχικοποίησή τους (σταθερές).

π.χ.
```
private final int x = 5;
x = 6;  // → ΛΑΘΟΣ!
```

β) Σε μεθόδους: Δεν γίνονται override (δεν υπαρκαλύπτονται).

π.χ.
```
public final void specialMethod()
{
  ...
}
```

γ) Σε κλάσεις: Δεν κληρονομούνται.

π.χ.
```
public final class MyClass
{
  ...
}
------------------
public class ClassName extends MyClass   // → ΛΑΘΟΣ!
```

Abstract κλάσεις και μέθοδοι:

Όπως αναφέρθηκε παραπάνω, σε μία δομή κληρονομικότητας, οι κοινές (πιο γενικές) ιδιότητες και συμπεριφορές ορίζονται όσο πιο ψηλά γίνεται στην ιεραρχία των κλάσεων. Όταν μία τέτοια ιδιότητα ή συμπεριφορά είναι μεν κοινή για διάφορες κλάσεις αλλά όχι σε τέτοιον βαθμό ώστε να μπορεί να οριστεί με ακρίβεια σε κάποιο υψηλό επίπεδο ιεραρχίας, τότε μπορεί να οριστεί σε μία abstract (αφηρημένη) κλάση με κάποια abstract μέθοδο. Όταν μία κλάση περιέχει έστω και μία abstract μέθοδο, θα πρέπει να ορισθεί ως abstract κλάση, ως εξής:

```
public abstract class <Όνομα_Κλάσης>
{
   ...
```

Μία abstract μέθοδος δεν έχει σώμα και άγγιστρα, δηλαδή έχει μόνο τον ορισμό της:

```
public abstract <τύπος> <όνομα_μεθόδου>;
```

Ουσιαστικά η μέθοδος ορίζει την ύπαρξή της μέσα σε μία abstract κλάση και αντιπροσωπεύει κάτι γενικό το οποίο θα ορισθεί σε χαμηλότερο επίπεδο ιεραρχίας, από κάποιες κλάσεις που θα κληρονομούν την abstract αυτή κλάση. Έτσι, η υλοποίησή της μπορεί να έχει διαφορετικές (παραπλήσιες συνήθως) μορφές. Κάποιες ιδιότητες των abstract κλάσεων:

- Δεν μπορεί να δημιουργηθεί αντικείμενο μιας abstract κλάσης
- Μια abstract κλάση μπορεί να έχει τις δικές της μεταβλητές όπως και static πεδία και μεθόδους
- Μια abstract κλάση μπορεί να έχει υποκλάσεις
  - Αν η υποκλάση μιας abstract κλάσης υλοποιεί όλες τις abstract μεθόδους της υπερκλάσης, τότε είναι μια κανονική κλάση
  - Αν η υποκλάση μιας abstract κλάσης δεν υλοποιεί όλες τις abstract μεθόδους της υπερκλάσης ή έχει κάποια δική της abstract μέθοδο, τότε και η υποκλάση προσδιορίζεται ως abstract

Παράδειγμα: Η abstract κλάση `Shape` που περιγράφει γενικά κάποιο σχήμα, περιέχει μια abstract μέθοδο `area` η οποία θα υλοποιηθεί σε κλάσεις που κληρονομούν τη `Shape` για να ορίσουν τον υπολογισμό του εμβαδού τού κάθε σχήματος. Κάθε σχήμα έχει ένα εμβαδόν επιφάνειας, όμως ο υπολογισμός του διαφέρει από σχήμα σε σχήμα, άρα η υλοποίηση της μεθόδου αυτής δε θα μπορούσε να γίνει στη γενική κλάση `Shape`.

```
public abstract class Shape
{
   private double x, y;

   public void location(double x, double y)
   {
      this.x = x;
      this.y = y;
   }

   public double getX()
   {
      return x;
   }
```

```
   public double getY()
   {
      return y;
   }

   public abstract double area();

   public String toString()
   {
      return "Σχήμα(x=" + x + ", y=" + y + ", εμβαδόν=" + area() + ")";
   }
}
```

Ακολουθούν δύο κλάσεις που κληρονομούν την κλάση `Shape` και οι οποίες υλοποιούν την abstract μέθοδο `area`:

```
public class Rectangle extends Shape
{
   private double height, width;

   public void setSize(double h, double w)
   {
      height = h;
      width = w;
   }

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

   public String toString()
   {
     return super.toString()+": Ορθογώνιο("+height+" x "+width+")";
   }
}

public class Circle extends Shape
{
   private double radius;

   public void setRadius(double r)
   {
      radius = r;
   }

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

   public String toString()
   {
      return super.toString() + ": Κύκλος(" + radius + ")";
   }
}
```

Στην κλάση εφαρμογής θα μπορούσε να υπάρχει κάτι τέτοιο:

```
public class TestAbstr
{
   public static void main(String [] args)
   {
      Rectangle r = new Rectangle();
      Circle c = new Circle();

      Shape [] s = new Shape[2];
      s[0] = r; s[1] = c;

      r.location(2, 3);
      r.setSize(40, 10);
      c.location(8, 4);
      c.setRadius(10);

      for (int i=0; i<s.length; i++)
         System.out.println(s[i]);
   }
}
```

ή

```
   for (Shape sh : s)
      System.out.println(sh);
```

Η έξοδος του προγράμματος είναι η εξής:

```
Σχήμα(x=2.0, y=3.0, εμβαδόν=400.0): Ορθογώνιο(40.0 x 10.0)
Σχήμα(x=8.0, y=4.0, εμβαδόν=314.1592653589793): Κύκλος(10.0)
```

Το πλεονέκτημα που φαίνεται στο συγκεκριμένο παράδειγμα είναι ότι η `toString` της `Shape` (η οποία κάνει χρήση της μεθόδου `area`) μπορεί να υλοποιηθεί χωρίς να έχει υλοποιηθεί ακόμα η μέθοδος `area`, κάνοντας χρήση μια γενικής (αφηρημένης) μορφής της `area`. Η συγκεκριμένη `toString` καλείται κάθε φορά μέσα από διαφορετική υποκλάση της `Shape`, χρησιμοποιώντας κάθε φορά διαφορετική υλοποίηση της `area`. Άρα, τα πλεονεκτήματα της κληρονομικότητας μπορούν να αξιοποιηθούν ακόμα και σε περιπτώσεις που κάποια γενική ιδιότητα (εδώ, ο υπολογισμός εμβαδού σχήματος) δεν μπορεί να υλοποιηθεί στη γενική κλάση που ανήκει (στην κλάση σχήματος, δηλ. στη `Shape`). Στο συγκεκριμένο παράδειγμα, μπορεί να μην αποφεύχθηκε η δημιουργία μεθόδου `area` για τον υπολογισμό του εμβαδού *σε κάθε κλάση* συγκεκριμένου σχήματος, αλλά αποφεύχθηκε η επανάληψη κώδικα που χρησιμοποιεί τον υπολογισμό των εμβαδών (δηλαδή, η επανάληψη του κώδικα της `toString` της `Shape`).

Προσοχή στο ότι στο παραπάνω πρόγραμμα δημιουργείται ένας "πίνακας αντικειμένων" της abstract κλάσης `Shape` (πίνακας τύπου `Shape`), όχι όμως και κανονικά αντικείμενα της `Shape`, τα οποία δεν θα μπορούσαν να δημιουργηθούν, αφού η κλάση `Shape` είναι abstract κλάση (δηλαδή, π.χ., η εντολή `Shape sh = new Shape();` θα έβγαζε σφάλμα). Το πρώτο στοιχείο του πίνακα `s` περιέχει το αντικείμενο της `Rectangle` και το δεύτερο περιέχει το αντικείμενο της `Circle`. Αυτό είναι εφικτό επειδή οι κλάσεις αυτές κληρονομούν την κλάση `Shape` (βλ. σελ. 9 παραπάνω).

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 10

Exceptions handling (Χειρισμός εξαιρέσεων)

Exception: σφάλμα που προκλήθηκε κατά την εκτέλεση του προγράμματος και που μπορεί να αντιμετωπιστεί από τον προγραμματιστή.

Error: σοβαρό σφάλμα που οφείλεται σε πρόβλημα που δεν μπορεί να αντιμετωπιστεί (π.χ., out of memory, stack overflow, κτλ) [θεωρητικά μπορεί να αντιμετωπιστεί, αλλά πρακτικά δεν αντιμετωπίζεται, αφού η λειτουργία του προγράμματος μετά από ένα τέτοιο σφάλμα είναι συνήθως προβληματική].

Τα exceptions μπορούν να αντιμετωπιστούν, άρα χρειάζονται ειδικό χειρισμό (exceptions handling).

Η κλάση `Exception` ορίζει απλές συνθήκες λάθους που μπορεί να συναντήσει κάποιο πρόγραμμα. Αντί να αφήνουμε το πρόγραμμα να τερματιστεί, μπορούμε να γράψουμε κώδικα που να χειρίζεται τις εξαιρέσεις και να συνεχίζει την εκτέλεση του προγράμματος. Όταν συμβεί κάποιο σφάλμα στο πρόγραμμα, ο κώδικας που το ανακαλύπτει «μεταβιβάζει» (`throws` – ο αγγλικός όρος) μια εξαίρεση.

→ Αν «συλλάβουμε» (`catch`) την εξαίρεση, είναι δυνατόν να επανακάμψουμε το πρόγραμμα ή να δώσουμε στον χρήστη κατάλληλα μηνύματα/οδηγίες. Η «σύλληψη» γίνεται εγκλωβίζοντας τον κώδικα που μπορεί να πετάξει/μεταβιβάσει μια εξαίρεση σε ένα `try`-block και τοποθετώντας στη συνέχεια ένα τουλάχιστον `catch`-block. Στο σημείο που θα συμβεί η εξαίρεση, η εκτέλεση του κώδικα διακόπτεται και η ροή του συνεχίζεται στο `catch`-block.

```
try
{
      // κώδικας που μπορεί να προκαλέσει μια συγκεκριμένη εξαίρεση
}
catch(ExceptionType e)
{
       // κώδικας που θα εκτελεστεί αν προκύψει η ExceptionType
}
```

Μπορεί να υπάρχουν και περισσότερα `catch` σε ένα `try`. Ο έλεγχος περνάει από το πρώτο στο επόμενο κτλ., μέχρι να βρεθεί το κατάλληλο. → Γι αυτό, ξεκινάμε από το πιο ειδικό (εξειδικευμένο) `catch` στο πιο γενικό.

Υπάρχει προαιρετικά και το block `finally` που περιέχει κώδικα ο οποίος θα εκτελεστεί είτε γίνει είτε δε γίνει κάποιο exception. Μάλιστα, ο κώδικας του `finally` θα εκτελεστεί ακόμα και εάν δε συλληφθεί κάποια εξαίρεση. Από τη στιγμή που ούτως ή άλλως ο κώδικας που υπάρχει μετά τα `catch` θα εκτελεστεί, η χρήση του `finally` επιβάλλεται κυρίως όταν το πρόγραμμα τερματίζεται νωρίτερα, π.χ. με κάποιο `return` ή μετά από κάποιο σφάλμα εξαίρεσης που δεν συλλαμβάνεται σε αντίστοιχο `catch`, όπως προαναφέρθηκε. Σε μια τέτοια περίπτωση, θα εκτελεστεί το `finally`-block και μετά θα τερματιστεί η εκτέλεση του προγράμματος.

- Εκτός από τις εξαιρέσεις που προκύπτουν αυτόματα όταν κάτι "πάει στραβά" σε κάποιο πρόγραμμα, ο προγραμματιστής έχει τη δυνατότητα να προκαλέσει ο ίδιος κάποια εξαίρεση. Ο τρόπος είναι ο εξής:

```
throw new ExceptionType("κείμενο περιγραφής της εξαίρ.");
```

Συνήθως μεταβιβάζουμε (προκαλούμε) κάποια εξαίρεση εάν συμβεί κάτι συγκεκριμένο, *δηλαδή συνήθως το `throw` βρίσκεται μέσα σε κάποια συνθήκη (π.χ., `if`)*. Στην ουσία, με το `throw new` δημιουργείται ένα αντικείμενο της κλάσης `ExceptionType` (του παραπάνω παραδείγματος) το οποίο παίρνει όνομα στο αντίστοιχο `catch`-block.

Άρα, γενικά, το ολοκληρωμένο block χειρισμού είναι: *`try-throw-catch`*:

```
try
{
   // δηλώσεις
   throw new ExceptionTypeX("κείμενο περιγραφής εξαίρεσης");
   // δηλώσεις
}
catch(ExceptionType1 e1)
{
   // κάνε αυτό αν συμβεί η εξαίρεση ExceptionType1
}
...

catch(ExceptionTypeX eX)
{
   // κάνε αυτό αν συμβεί η εξαίρεση ExceptionTypeX
}
...

[finally
{
      // κάνε αυτό είτε συμβεί είτε δε συμβεί κάποια
      // από τις παραπάνω εξαιρέσεις
}]
```

- Τα `ExceptionTypeX` είναι συγκεκριμένοι τύποι εξαιρέσεων της Java, π.χ., `ArrayIndexOutOfBoundsException`, `NullPointerException`, `IOException`, `ClassNotFoundException`, `FileNotFoundException`, κτλ.

- Υπάρχουν μέθοδοι που επιστρέφουν περιγραφή της κάθε εξαίρεσης, π.χ.:

```
try
{
    ...
}
catch(ArrayIndexOutOfBoundsException e)
{
   System.out.println(e.toString()); // σύντομη περιγραφή εξαίρεσης
   System.out.println(e.getMessage()); // αναλυτική περιγρ. εξαίρεσης
}
```

Προσοχή! Αν μια εξαίρεση δεν συμβαίνει μέσα σε `try {...}` ή αν δεν συλλαμβάνεται στη συνέχεια σε κάποιο `catch(...){ }`, είναι σαν να μην υπάρχει, από άποψη χειρισμού (handling).

Παράδειγμα "try-throw-catch"

```
import java.util.*;

public class ExceptionsExample1
{
   public static void main(String [] args)
   {
      int tyropites, mathites;
      double tyropitesAnaMathiti;
      Scanner input = new Scanner(System.in);
      try
      {
         System.out.println("Πόσες τυρόπιτες?");
         tyropites = input.nextInt();
         System.out.println("Πόσοι μαθητές?");
         mathites = input.nextInt();
         if (mathites < 1)
            throw new Exception("Εξαίρεση: Δεν υπάρχουν μαθητές!");
         tyropitesAnaMathiti = tyropites / (double)mathites;
         System.out.println("Τυρόπιτες ανά μαθητή = " + tyropitesAnaMathiti);
      }
      catch (Exception e)
      {
         System.out.println(e.getMessage());
      }
      System.out.println("Τέλος προγράμματος.");
   }
}
```

Με inputs 15 και 10:

→ 15
→ 10
Τυρόπιτες ανά μαθητή: 1.5
Τέλος προγράμματος.

Με inputs 15 και 0:

→ 15
→ 0
Εξαίρεση: Δεν υπάρχουν μαθητές!
Τέλος προγράμματος.

Στη δεύτερη εκτέλεση του προγράμματος (με δεδομένα 15 και 0), δεν εκτελείται η εντολή `System.out.println("Τυρόπιτες ανά μαθητή = " + tyropitesAnaMathiti);` αφού στην ακριβώς προηγούμενη εντολή προκύπτει εξαίρεση, οπότε σταματάει η κανονική ροή του κώδικα και εκτελείται η εντολή του `catch`-block.

Δημιουργία κλάσης εξαίρεσης

Οι εξαιρέσεις είναι υποκλάσεις της κλάσης `Exception`. Άρα, όταν θέλουμε να δημιουργήσουμε μία δική μας κλάση εξαίρεσης, θα πρέπει να κληρονομούμε την κλάση `Exception`. Π.χ.:

```
public class DivideByZeroException extends Exception
{
   public DivideByZeroException()
   {
      super("Dividing by zero!");
   }
```

```
   public DivideByZeroException(String message)
   {
      super(message);
   }
}
```

και μία κλάση που χρησιμοποιεί αυτή την κλάση εξαίρεσης:

```
import java.util.*;
public class DivByZeroExample
{
   private int num, den;
   private double fraction;

   public void divide()
   {
      try
      {
         System.out.println("Αριθμητής?");
         Scanner input = new Scanner(System.in);
         num = input.nextInt();
         System.out.println("Παρανομαστής?");
         den = input.nextInt();
         if (den == 0)
             throw new DivideByZeroException();
         fraction = num / (double)den;
         System.out.println("Κλάσμα = " + fraction);
      }
      catch(DivideByZeroException e)
      {
         System.out.println(e.getMessage());
         secondChance();
      // υποθέτουμε ότι υπάρχει και μία μέθοδος secondChance
      // που ξαναζητάει την πιο προσεχτική είσοδο δεδομένων...
      }
   } // end of method devide

   public static void main(String [ ] args)
   {
      DivByZeroExample obj = new DivByZeroExample();
      obj.divide();
   }
}
```

[Στο συγκεκριμένο παράδειγμα, η μέθοδος `main` βρίσκεται στην ίδια την κλάση που βρίσκεται και η `divide`. Επειδή η `main` είναι `static`, δεν έχει άμεση πρόσβαση στη μέθοδο `divide` που δεν είναι `static`, οπότε για να την καλέσει πρέπει πρώτα να δημιουργήσει αντικείμενο (`obj`) της ίδιας της κλάσης στην οποία βρίσκεται. Αυτή η δημιουργία, όπως έχει αναφερθεί και σε προηγούμενο κεφάλαιο, είναι η μόνη λύση για την κλήση μη-`static` μεθόδων από `static` μεθόδους της ίδιας κλάσης. Εναλλακτικά, θα μπορούσε προφανώς να δηλωθεί ως `static` η μέθοδος `divide`.]

Δήλωση εξαιρέσεων σε μέθοδο:

Υπάρχουν περιπτώσεις που κάποια εξαίρεση μπορεί να θέλουμε να την κάνουμε `catch` μόνο εάν συντρέχουν κάποιοι λόγοι, που σημαίνει ότι μπορεί και να μην θέλουμε να την κάνουμε `catch` (δηλαδή να μη θέλουμε να τη διαχειριστούμε και να αφήσουμε το πρόγραμμα να τερματιστεί). Σε αυτές τις περιπτώσεις, ο κώδικας που μπορεί να μεταβιβάσει εξαίρεση μπαίνει *σε μια μέθοδο* (η

οποία στην ουσία *μεταβιβάζει (μεταθέτει) την ευθύνη του χειρισμού της εξαίρεσης* στη μέθοδο που την καλεί) *η οποία δεν περιέχει* `try-catch` *block*, αλλά:

- *δηλώνει* στον ορισμό της (προειδοποιεί) ότι πιθανόν θα προκαλέσει κάποια εξαίρεση → αυτό γίνεται με την *πρόταση* `throws`
- *καλείται* μέσα σε ένα `try`-block (σε μια άλλη μέθοδο προφανώς) το οποίο φυσικά έχει και το αντίστοιχο `catch` (εάν τελικά θέλουμε να διαχειριστούμε την εξαίρεση – διαφορετικά η μέθοδος καλείται χωρίς `try-catch`)

Παράδειγμα:

```
import java.util.*;
public class ExceptionsExample2
{
   Scanner input = new Scanner(System.in);
   public int getInt() throws Exception
   {
      System.out.println("Πληκτρολόγησε έναν ακέραιο:");
      int n = input.nextInt();
      return n;
   }

   public void divide()
   {
      int n1=0, n2=1, n3=0;
      try
      {
         n1 = getInt();
         n2 = getInt();
         n3 = n1/n2;
      }
      catch(Exception e)
      {
         System.out.println(e.toString());
      }
      System.out.println(n1 + "/" + n2 + "=" + n3);
   }
}
```

Η μέθοδος `divide()` καλείται από τη `main` μέθοδο, στην κλάση εφαρμογής. Στη `main` λοιπόν:

```
ExceptionsExample2 ex2 = new ExceptionsExample2();
ex2.divide();
```

Αν τα inputs είναι π.χ. 22 και 3, τότε το πρόγραμμα θα τυπώσει:

```
22/3=7
```

Αν όμως τα inputs είναι π.χ. 22 και w, θα τυπώσει:

```
[InputMismatchException]
22/1=0
```

ενώ αν τα inputs είναι 22 και 0, τότε θα τυπώσει:

```
[ArithmeticException: / by zero]
22/0=0
```

Στη δεύτερη περίπτωση, ο κώδικας μεταβιβάζει εξαίρεση γιατί αντί για ακέραιος δόθηκε χαρακτήρας (w) για το `n2`. Η εξαίρεση προκαλείται κατά την κλήση της μεθόδου `getInt()`, η οποία όντως προειδοποιούσε ότι μπορεί να συνέβαινε κάτι τέτοιο, και συλλαμβάνεται από το `catch`, το οποίο τυπώνει το σύντομο μήνυμα. Ο κώδικας συνεχίζει να εκτελείται, οπότε τυπώνονται οι τιμές των `n1`, `n2` και `n3`. Η τιμή του `n1` είναι αυτή που δόθηκε από τον χρήστη, η τιμή του `n2` (1) είναι η αρχική του τιμή, αφού η εντολή απόδοσης τιμής στη μεταβλητή αυτή δεν εκτελέστηκε επειδή συνέβη η εξαίρεση, και το ίδιο ισχύει και για το `n3` (όταν συμβαίνει κάποιο exception, η ροή του κώδικα διακόπτεται και πηγαίνει στο αντίστοιχο `catch`).

Αντίστοιχα, στην τρίτη περίπτωση, επιχειρείται να γίνει διαίρεση με το 0, οπότε ο κώδικας μεταβιβάζει την ανάλογη εξαίρεση (η εξαίρεση πλέον προκαλείται από εντολή της μεθόδου `divide()` και όχι από μεταβίβαση από τη μέθοδο `getInt()`), η οποία συλλαμβάνεται από το `catch`, το οποίο τυπώνει το σύντομο μήνυμα. Ο κώδικας συνεχίζει να εκτελείται, οπότε τυπώνονται οι τιμές που δόθηκαν για τα `n1` και `n2`, ενώ η τιμή 0 του `n3` είναι η αρχική του τιμή, αφού στην ουσία η διαίρεση δεν έγινε ποτέ.

Στο συγκεκριμένο παράδειγμα, το `catch` είναι πολύ γενικό για να συλλάβει και τις δύο διαφορετικές εξαιρέσεις που συμβαίνουν κατά την εκτέλεση.

Γενικός κανόνας: Αν μία μέθοδος μεταβιβάζει κάποια εξαίρεση (με την εντολή `throw`), τότε είτε πρέπει να περιέχει κάποιο `catch` για να τη συλλάβει, είτε πρέπει να δηλώνει στον ορισμό της ότι μεταβιβάζει κάποια εξαίρεση (με το `throws`).

Παράδειγμα:

```
public class CatchExample
{
   public static void main(String [] args)
   {
      CatchExample obj = new CatchExample();

      try
      {
         System.out.println("Trying");
         obj.myMethod();
         System.out.println("Trying after call.");
      }
      catch(Exception e)
      {
         System.out.println("Catching");
         System.out.println(e.getMessage());
      }
   }

   public void myMethod() throws Exception
   {
      System.out.println("Starting myMethod.");
      throw new Exception("From myMethod.");
   }
}
```

Το πρόγραμμα αυτό θα εκτυπώσει:

```
Trying
Starting myMethod.
Catching
From myMethod.
```

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 11

Η κλάση ArrayList

Οι λίστες ArrayList είναι δυναμικές δομές δεδομένων.
Κυριότερο χαρακτηριστικό τους: έχουν δυναμικό (μεταβαλλόμενο) μέγεθος.

Το μέγεθος ενός πίνακα δε μπορεί να αλλάξει από τη στιγμή που ο πίνακας οριστεί, ακόμα και αν το μέγεθος δεν ορίζεται επ'ακριβώς μέσα στο πρόγραμμα αλλά το δίνει ο χρήστης ως είσοδο στο πρόγραμμα.

π.χ.
```
private double [ ] a;
...
public void createArray(int size)
{
      a = new double [size];
}
```

δηλαδή αρχικά το μέγεθος του πίνακα `a` δεν είναι γνωστό, αλλά όταν κληθεί η μέθοδος `createArray`, το μέγεθος θα είναι γνωστό και θα σταλεί ως όρισμα από τον χρήστη κατά την εντολή κλήσης της μεθόδου, οπότε ο πίνακας δημιουργείται μέσα στη μέθοδο. Αν κατόπιν θέλουμε να έχουμε περισσότερα στοιχεία (elements) στον `a`, τότε υπάρχει πρόβλημα! Για αυτόν τον λόγο υπάρχουν οι λίστες ArrayList, που έχουν μεταβλητό μέγεθος.

→ Τότε γιατί να μην χρησιμοποιούμε πάντα λίστες ArrayList αντί για πίνακες;
Οι λίστες ArrayList έχουν τρία βασικά μειονεκτήματα:

- Είναι λιγότερο αποδοτικές από τους πίνακες,
- Είναι πιο δύσχρηστες από τους πίνακες γιατί δεν υποστηρίζουν τη χρήση αγκυλών `[ ]`
- Τα στοιχεία τους πρέπει να είναι αντικείμενα (objects). Δε μπορούν να είναι απλές τιμές πρωτογενών μεταβλητών (int, double, char, κτλ.) όπως στους πίνακες (αυτό φυσικά έχει και πάρα πολλά πλεονεκτήματα).

Χρήση των λιστών ArrayList:

Ο ορισμός της κλάσης `ArrayList` δεν παρέχεται αυτόματα. Βρίσκεται στο πακέτο `java.util` , το οποίο πρέπει να γίνει `import` από οποιοδήποτε πρόγραμμα χρησιμοποιεί λίστες ArrayList:

```
import java.util.*;
```

Δημιουργία λίστας ArrayList (αντικειμένου της ArrayList):

Η `ArrayList` είναι κλάση, άρα οι λίστες δημιουργούνται όπως δημιουργούνται τα αντικείμενα:

```
ArrayList listName = new ArrayList();
```

Κάθε λίστα ArrayList έχει μια χωρητικότητα (capacity) και ένα μέγεθος (size). Η χωρητικότητα

ορίζει το πλήθος των στοιχείων που μπορεί να χωρέσει μία λίστα ArrayList. Μέγεθος είναι το πλήθος των στοιχείων που έχει ανά πάσα στιγμή η λίστα ArrayList. Η χωρητικότητα μιας λίστας ArrayList είναι πάντα μεγαλύτερη ή ίση με το μέγεθός της (capacity >= size) και αυξάνεται αυτόματα κάθε φορά που το μέγεθος πάει να την ξεπεράσει, δηλαδή όταν προστεθεί ένα επιπλέον στοιχείο σε μία "γεμάτη" λίστα.

Προαπαιτούμενα για τη χρήση λιστών ArrayList:

Generics

Από την έκδοση 5.0 της Java και μετά, υπάρχει η δυνατότητα ορισμού κλάσεων ή μεθόδων που να έχουν ως παραμέτρους **τύπους δεδομένων**. Η διαφορά με τις κανονικές παραμέτρους εισόδου που γνωρίζαμε ως τώρα στους ορισμούς των μεθόδων, είναι ότι οι κανονικές παράμετροι δέχονται ως είσοδο τιμές, ενώ οι παράμετροι τύπου δέχονται ως είσοδο τύπους δεδομένων.

Οι κλάσεις που ορίζονται με τη χρήση τύπων δεδομένων ως παραμέτρους (generics) ονομάζονται generic classes (γενικευμένες κλάσεις) και ορίζονται ως εξής:

```
public class Όνομα_Κλάσης<Τύπος_δεδομένων1, Τύπος_δεδομένων_2, ...>
{
    ...
}
```

Όταν μία κλάση είναι generic, τότε κατά τη δημιουργία αντικειμένου της ορίζεται κάθε φορά ο παραμετρικά οριζόμενος τύπος της (ή οι τύποι της, αν είναι πολλοί) με συγκεκριμένο τύπο κλάσης, ως εξής:

```
Κλάση<τύπος_κλάσης_1, τύπος_κλάσης_2, ...> αντικείμενο =
    new  Κλάση<τύπος_κλάσης_1, τύπος_κλάσης_2, ...>();
```

Παράδειγμα:

Δημιουργία μιας generic κλάσης `Item`:

```
public class Item<Key,Value>
{
   private Key k;   // δήλωση μιας μεταβλητής generic τύπου Key
   private Value v; // δήλωση μιας μεταβλητής generic τύπου Value

   // κατασκευαστής:
   public Item(Key a, Value b)
   {
      k = a;
      v = b;
   }

   public void put(Key kk, Value vv)
   {
      k = kk;
      v = vv;
   }
```

```
    public Value get()
    {
        return v;
    }
}
```

Η γενικευμένη αυτή κλάση έχει δύο παραμέτρους τύπου δεδομένων. Οι παράμετροι τύπου δεδομένων μιας κλάσης (εδώ `Key` και `Value`) χρησιμοποιούνται μέσα στην κλάση σαν να είναι κανονικοί τύποι κλάσης (θα γίνουν συγκεκριμένοι τύποι κατά τη δημιουργία κάποιου αντικειμένου της `Item`, με τη δυνατότητα να είναι διαφορετικού τύπου για κάθε αντικείμενο). Έτσι, η κλάση αυτή μπορεί να χρησιμοποιηθεί με διάφορους τρόπους, πχ.

```
public static void main(String [] args)
{
// Δημιουργία αντικειμένου της Item που αντιπροσωπεύει στοιχείο τηλεφωνικού
// καταλόγου (όνομα και τηλέφωνο):
  Item<String, Integer> phonebook = new Item<String, Integer>("Babis", 3456789);

// Δημιουργία αντικειμένου της Item που αντιπροσωπεύει στοιχείο τιμοκαταλόγου
// προϊόντων (κωδικός προϊόντος και τιμή):
  Item<Integer, Double> product = new Item<Integer, Double>(1001, 449.99);

  System.out.println(phonebook.get());  // εμφάνιση τηλεφωνικού αριθμού
  System.out.println(product.get());    // εμφάνιση τιμής
}
```

Αυτόματη συσκευασία – αποσυσκευασία (Autoboxing – unboxing)

Σε αρκετές περιπτώσεις μέχρι τώρα έχουμε χρησιμοποιήσει αντικείμενα των κλάσεων `Integer` και `Double`. Οι κλάσεις αυτές, μαζί με τις κλάσεις `Character` και `Boolean`, ονομάζονται **wrapper κλάσεις** και τα αντικείμενά τους κρατούν τιμές των αντίστοιχων πρωτογενών τύπων. Χρησιμοποιούνται κυρίως σε περιπτώσεις που απαιτείται η χρήση αντικειμένου, όμως η τιμή τού αντικειμένου είναι ουσιαστικά πρωτογενούς τύπου.

Από την έκδοση 5.0 της Java και μετά, υποστηρίζεται η αυτόματη μετατροπή πρωτογενών τύπων στα αντικείμενα των αντίστοιχων wrapper κλάσεών τους, σε εκφράσεις και κλήσεις μεθόδων. Η μετατροπή αυτή ονομάζεται *αυτόματη συσκευασία* (autoboxing). Η Java 5.0 και οι επόμενες εκδόσεις της υποστηρίζουν επίσης την αυτόματη αποσυσκευασία (automatic unboxing), όπου αντικείμενα wrapper κλάσεων μετατρέπονται αυτόματα στους αντίστοιχους πρωτογενείς τύπους όταν είναι απαραίτητο σε κάποια εκχώρηση ή κλήση μεθόδου. Π.χ., η κανονική δημιουργία του `Integer i1` με τιμή 5:

```
Integer i1 = new Integer(5);
```

θα μπορούσε να γίνει και ως εξής:

```
Integer i1 = 5;    // autoboxing
```

Αντίστροφα, η ακόλουθη εκχώρηση είναι σωστή, αφού πραγματοποιείται αυτόματη μετατροπή του `Integer i1` σε `int`:

```
int i2 = i1;   // auto unboxing
```

Πιο συγκεκριμένα παραδείγματα κατά την κλήση μεθόδων θα φανούν παρακάτω, κατά την παρουσίαση κάποιων μεθόδων της κλάσης `ArrayList`.

Κατασκευαστές της `ArrayList`:

Η κλάση `ArrayList` είναι μια generic κλάση, επομένως κατά τη δημιουργία αντικειμένων της, δηλαδή κάποιας λίστας ArrayList, μπορεί να οριστεί επακριβώς ο τύπος των αντικειμένων που θα περιέχει η λίστα. (Όπως αναφέρθηκε στην αρχή, τα στοιχεία μιας λίστας ArrayList είναι αντικείμενα). Ο τύπος αυτός λέγεται **βασικός τύπος (base type)** της λίστας. Με τον τρόπο αυτόν, ο compiler ελέγχει κατά τη μεταγλώττιση εάν στη λίστα εισάγονται αντικείμενα του συγκεκριμένου τύπου και δεν επιτρέπει την εισαγωγή αντικειμένων διαφορετικού τύπου. Ως βασικός τύπος κάποιας λίστας ArrayList μπορεί να οριστεί οποιοσδήποτε τύπος κλάσης, όχι όμως κάποιος πρωτογενής τύπος. Όταν θέλουμε να αποθηκεύσουμε πρωτογενείς τύπους μεταβλητών σε μια λίστα ArrayList, χρησιμοποιούμε τις αντίστοιχες wrapper κλάσεις ως βασικούς τύπους (`Integer` για `int`, `Double` για `double` κτλ.)

Οι κατασκευαστές της κλάσης `ArrayList` είναι οι ακόλουθοι:

- `public ArrayList<Base_Type>()` – δημιουργεί μία κενή λίστα ArrayList με στοιχεία βασικού τύπου Base_Type, με αρχική χωρητικότητα = 10. Κάθε φορά που χρειάζεται αύξηση της χωρητικότητας, αυτό γίνεται αυτόματα.

- `public ArrayList<Base_Type>(int initialCapacity)` – όπως προηγουμένως, αλλά η αρχική χωρητικότητα είναι = `initialCapacity`.

  π.χ., `ArrayList<Integer> list1 = new ArrayList<Integer>(35);`

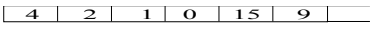
Μέθοδοι της `ArrayList`:

- Προσθήκη στοιχείου:

- στο τέλος της λίστας ArrayList (μετά το τελευταίο στοιχείο):

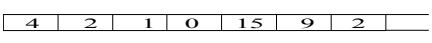
```
public boolean add(Base_Type newElement)
```

Ας υποθέσουμε ότι έχουμε τη λίστα ArrayList `list2`:



της οποίας τα στοιχεία είναι τύπου `Integer` (το Base_Type), και θέλουμε να προσθέσουμε μια `int` τιμή στο τέλος της λίστας. Αυτό μπορεί να γίνει ως εξής:

```
private int x = 2;
list2.add(x); →
```



Η μέθοδος `add` επιστρέφει `true` εάν λειτουργήσει με επιτυχία. Συνήθως χρησιμοποιείται χωρίς να εκχωρείται σε κάποια μεταβλητή η κλήση της (όπως στο παράδειγμα παραπάνω), δηλαδή σαν να ήταν `void`.

Κανονικά, η κλήση της `add` στο παραπάνω παράδειγμα θα έπρεπε να ήταν ως εξής:

```
list2.add(new Integer(x));
```

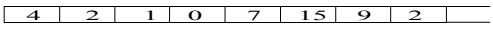
όμως αυτό δεν είναι απαραίτητο, αφού κατά την εκτέλεση του `list2.add(x)` πραγματοποιείται *αυτόματη συσκευασία (autoboxing)* και ο ακέραιος 2 μετατρέπεται σε αντικείμενο της wrapper κλάσης `Integer` με τιμή 2.

- ενδιάμεσα:

```
public void add(int index, Base_Type newElement)
```

Η υπερφορτωμένη αυτή έκδοση της `add` προσθέτει το νέο στοιχείο σε μια ενδιάμεση θέση της λίστας (θέση index). Τα στοιχεία που βρίσκονταν από τη θέση index και μετά, μετακινούνται κατά μία θέση δεξιά.

π.χ., `list2.add(4, 7);` →



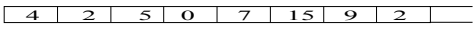

```
// ή: list2.add(4, new Integer(7));
```

- **<u>Αντικατάσταση (υπάρχοντος) στοιχείου:</u>**

```
public Base_Type set(int index, Base_Type newElement)
```

π.χ., θέλουμε να αλλάξουμε το *τρίτο* στοιχείο της `list2` σε "5":

`list2.set(2, 5);` →



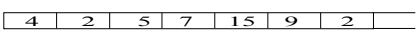
Η μέθοδος `set` επιστρέφει την τιμή του στοιχείου που βρισκόταν πριν σε εκείνη τη θέση, όμως, όπως και η `add`, συνήθως χρησιμοποιείται χωρίς να εκχωρείται σε κάποια μεταβλητή η κλήση της (όπως στο παράδειγμα παραπάνω), δηλαδή σαν να ήταν `void`.

- **<u>Διαγραφή στοιχείου/ων:</u>**

- `public Base_Type remove(int index)`

Αφαιρεί από τη λίστα ArrayList το στοιχείο που βρίσκεται στη θέση index. Επιστρέφει το στοιχείο που αφαιρέθηκε, αλλά συνήθως χρησιμοποιείται και αυτή σαν να ήταν `void`.

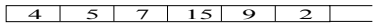
π.χ., `list2.remove(3);` →



- `public boolean remove(Object theElement)`

Διαγράφει το στοιχείο theElement *μόνο από την πρώτη του θέση* στη λίστα. Επιστρέφει `true` εάν βρει το στοιχείο και το διαγράψει, και `false` εάν δεν το βρει. Τα στοιχεία που βρίσκονταν δεξιά του στοιχείου που διαγράφηκε, μετακινούνται κατά μία θέση αριστερά.

π.χ., `list2.remove(2);` → `true` →



`list2.remove(3);` → `false`

- `protected void` **`removeRange`**`(int fromIndex, int toIndex)`

Διαγράφει όλα τα στοιχεία της λίστας ArrayList με δείκτες από fromIndex έως toIndex, μη συμπεριλαμβανομένου του στοιχείου στη θέση toIndex. Τα στοιχεία από τη θέση toIndex έως το τέλος της λίστας μεταφέρονται αριστερά ώστε να καλύψουν τις κενές θέσης που δημιουργήθηκαν.

- `public void` **`clear`**`()`

Διαγράφει όλα τα στοιχεία της λίστας ArrayList με την οποία καλείται και θέτει το μέγεθός της ίσο με το μηδέν.

- <u>Πρόσβαση σε στοιχείο:</u>

- `public Base_Type` **`get`**`(int index)`

Επιστρέφει το στοιχείο της λίστας ArrayList που βρίσκεται στη θέση index. Στις νέες εκδόσεις της Java που υπάρχει η αυτόματη αποσυσκευασία (auto-unboxing), δεν απαιτείται κάποια μετατροπή των στοιχείων που επιστρέφει η `get` όταν ο βασικός τύπος της λίστας είναι κάποια wrapper κλάση. Π.χ,

`int sum = list2.get(2) + list2.get(4);` → `7 + 9` → `sum = 16`

Αν όμως ο βασικός τύπος της λίστας ArrayList δεν είχε οριστεί, δηλαδή η `list2` είχε δημιουργηθεί έτσι:

```
ArrayList list2 = new ArrayList();
```

και όχι έτσι:

```
ArrayList<Integer> list2 = new ArrayList<Integer>();
```

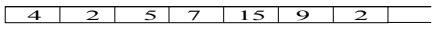
η παραπάνω εντολή εκχώρησης στην `int` μεταβλητή `sum` θα ήταν λάθος (σφάλμα κατά το compilation), και θα έπρεπε να γραφεί ως εξής (μετατρέποντας τα γενικά αντικείμενα που επιστρέφει η `get` σε αντικείμενα τύπου `Integer`, τα οποία μετά "αποσυσκευάζονται" αυτόματα σε `int`):

```
int sum = (Integer)list2.get(2) + (Integer)list2.get(4);
          |          |____Object___||
          |_____Integer object ______|
```

- <u>Μέθοδοι αναζήτησης:</u>

- `public boolean` **`contains`**`(Object target)` → `true/false`

π.χ., αν έχουμε το `list2` στη μορφή:




```
list2.contains(15);  → true
```

- `public int` **`indexOf`**`(Object target)` → 1ο index του target
ή -1 αν δεν υπάρχει

π.χ., πάντα για το ίδιο `list2`:

```
list2.indexOf(2);  → 1
list2.indexOf(3);  → -1
```

- `public int` **`lastIndexOf`**`(Object target)`

π.χ., `list2.lastIndexOf(2);` → 6

- <u>Άλλες μέθοδοι:</u>

- `public int` **`size`**`()` → επιστρέφει το τρέχον πλήθος των στοιχείων της λίστας (μέγεθος)

- `public void` **`trimToSize`**`()` → κάνει το capacity = με το size

- `public Object []` **`toArray`**`()` → επιστρέφει πίνακα που περιέχει όλα τα στοιχεία της λίστας

Π.χ.,

```
Object [] z = list2.toArray();
for (Object x : z)
   System.out.print(x + " ");          → 4 2 5 7 15 9 2
```

Προφανώς, η άμεση εκτύπωση των στοιχείων της λίστας ArrayList `list2` γίνεται ως εξής:

```
for (Integer x : list2)
   System.out.print(x + " ");          → 4 2 5 7 15 9 2
```

<u>Παράδειγμα χρήσης λίστας ArrayList:</u>

- Να γραφεί μέθοδος που να ζητάει από τον χρήστη εισαγωγή double τιμών διαφορετικών του μηδενός μέχρι να δοθεί η τιμή 0 (για τερματισμό της διαδικασίας), και να αποθηκεύει τις τιμές αυτές.

Μπορούμε να χρησιμοποιήσουμε έναν `double` πίνακα;

Όχι, γιατί δεν ξέρουμε εξ αρχής πόσες τιμές θα εισάγει ο χρήστης! Άρα, χρησιμοποιούμε μια λίστα `ArrayList` για την αποθήκευση των τιμών. Ένας τρόπος είναι ο ακόλουθος (υπάρχουν φυσικά διάφοροι τρόποι για να έχουμε το ίδιο αποτέλεσμα):

```
import java.util.*;    // για χρήση της ArrayList

public class ArrayListExample
{
   ArrayList<Double> data = new ArrayList<Double>();

   Scanner input = new Scanner(System.in);  // για user input

   // Η μέθοδος που ζητείται:
   public void askAndStoreData()
   {
      double d;
      do
      {
          System.out.println("Δώσε έναν πραγματικό αριθμό ή 0 για τερματισμό.");

         d = input.nextDouble();

         data.add(d);
      } while (d != 0);  // επανέλαβε όσο η τιμή δεν γίνεται 0

      // πρέπει να σβήσουμε το 0 που αποθηκεύτηκε στο τέλος!
      data.remove(0.0);  // ΠΡΟΣΟΧΗ! Όχι: data.remove(0);

   } // end method

} // end class
```

Σημείωση: Αν η τελευταία εντολή ήταν `data.remove(0);` δεν θα πραγματοποιούνταν η επιθυμητή αφαίρεση του μηδενός από το τέλος της λίστας, αφού το στοιχείο 0 στη λίστα είναι τύπου `Double` και όχι `Integer`. Για την ακρίβεια, αυτό που θα γινόταν στη συγκεκριμένη περίπτωση, θα ήταν να κληθεί η έκδοση της μεθόδου `remove` που δέχεται ακέραιο (`public Base_Type remove(int index)`) και αφαιρεί το στοιχείο που βρίσκεται στη θέση τού συγκεκριμένου ακεραίου, οπότε θα αφαιρούνταν από τη λίστα το 1ο της στοιχείο.

ΠΛΗΡΟΦΟΡΙΚΗ ΙΙ (Java)
Ενότητα 12

Ροές και είσοδος/έξοδος αρχείων

Τα *δεδομένα εισόδου* ενός προγράμματος, εκτός από το να δίνονται από τον χρήστη με τη χρήση του πληκτρολογίου και/ή παραθυρικού περιβάλλοντος, μπορούν να δίνονται στο πρόγραμμα μέσω ενός αρχείου στο οποίο βρίσκονται αποθηκευμένα. Ομοίως, τα *αποτελέσματα* ενός προγράμματος (δεδομένα εξόδου), εκτός από το να εμφανίζονται στην οθόνη (στο terminal window ή σε κάποιο παραθυρικό περιβάλλον), μπορούν να αποθηκεύονται σε κάποιο αρχείο, το οποίο μπορεί μετέπειτα να διαβάσει ο χρήστης π.χ. με έναν κειμενογράφο.

Η είσοδος/έξοδος δεδομένων από/σε αρχεία στη Java γίνεται μέσω *ροών (streams)* → αντικείμενα συγκεκριμένων κλάσεων για είσοδο/έξοδο.

→ Ροές χρησιμοποιούνται και για την έξοδο στην οθόνη ή την είσοδο από το πληκτρολόγιο. Π.χ., η `System.out` που χρησιμοποιείται για την έξοδο στο terminal window (π.χ. `System.out.println()`) είναι μια ροή εξόδου, ενώ τα αντικείμενα της `Scanner` που χρησιμοποιούνται για είσοδο από το πληκτρολόγιο είναι ροές εισόδου.

Δηλαδή: streams → input streams (π.χ., πληκτρολόγιο ή αρχείο)
streams → output streams (π.χ., οθόνη ή αρχείο)

Έξοδος σε αρχείο κειμένου (ροή χαρακτήρων)

Χρήση της κλάσης `FileWriter` του πακέτου `java.io`.
Άρα:
```
import java.io.*;
```

Διαδικασία εγγραφής σε αρχείο, σε 3 βήματα:

- Δημιουργία & άνοιγμα αρχείου (στην ουσία, δημιουργία ροής εξόδου):

```
FileWriter outputStream = new FileWriter("όνομα αρχείου");
```

- Εγγραφή χαρακτήρα στο αρχείο:

```
outputStream.write(χαρακτήρας);
```

- Κλείσιμο αρχείου:

```
outputStream.close();
```

Κάποιες παρατηρήσεις:

- Το αρχείο, αν δεν υπάρχει, δημιουργείται. Αν υπάρχει ήδη, τότε δημιουργείται εκ νέου, άρα το αρχικό αρχείο διαγράφεται (χάνεται).
- Για εγγραφή δεδομένων στο τέλος υπάρχοντος αρχείου (χωρίς δηλαδή να χαθούν τα

δεδομένα που έχει ήδη το αρχείο):

```
FileWriter outputStream = new FileWriter("όνομα αρχείου", true);
```

Αν ως δεύτερο όρισμα εισόδου στον κατασκευαστή της `FileOutputStream` δοθεί `boolean` μεταβλητή με τιμή `false`, τότε η ροή εξόδου λειτουργεί όπως πριν, δηλαδή σαν να μην υπήρχε δεύτερο όρισμα, άρα διαγράφεται το αρχείο σε περίπτωση που υπάρχει ήδη.

- Αφού ολοκληρωθεί η εγγραφή δεδομένων στο αρχείο, θα πρέπει πάντα να εκτελείται η εντολή που κλείνει το αρχείο, ώστε να διατηρηθεί η γενικότερη ευστάθεια του (λειτουργικού) συστήματος.

- Συνήθως, η δημιουργία της ροής της `FileWriter` γίνεται σε δύο στάδια, με χρήση try-block ώστε να μπορεί να γίνει χειρισμός πιθανής εξαίρεσης. Δηλαδή:

εκτός try:
```
FileWriter outputStream = null;
```
μέσα σε try:
```
try {
     outputStream = new FileWriter("όνομα");
    }
catch (IOException e) {
     ... }
```

Παράδειγμα εξόδου από αρχείο:

Εγγραφή των χαρακτήρων που αντιστοιχούν στους κώδικες ASCII 33-126 σε αρχείο με όνομα `ASCII.txt` με 16 χαρακτήρες ανά γραμμή χωρίς κενά μεταξύ τους.

```
import java.io.*;  // για χρήση αρχείων

public class PrintASCII
{
     public static void main(String args[])
     {
          FileWriter fw = null;
          try {
               fw = new FileWriter("ASCII.txt");

               for (int i=33; i<127; i++)
               {
                  fw.write(i);
                  if (i%16 == 0)
                     fw.write('\n'); // αλλαγή γραμμής
               }
               if (fw != null)
                  fw.close();
          }
          catch (IOException e) {
               e.printStackTrace();
          }
     }
}
```

Το περιεχόμενο του αρχείου `ASCII.txt` μετά την εκτέλεση του παραπάνω προγράμματος θα είναι

```
!"#$%&'()*+,-./0
123456789:;<=>?@
ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]^_`
abcdefghijklmnop
qrstuvwxyz{|}~
```

Είσοδος από αρχείο κειμένου (ροή χαρακτήρων)

Χρήση της κλάσης `FileReader` του πακέτου `java.io`. Άρα: `import java.io.*;`

Διαδικασία ανάγνωσης από αρχείο, σε 3 βήματα:

- Άνοιγμα αρχείου (στην ουσία, δημιουργία ροής εισόδου):

```
FileReader inputStream = new FileReader("όνομα αρχείου"));
```

- Ανάγνωση χαρακτήρα από το αρχείο: `inputStream.read();`

  Επιστρέφει τιμή `int` (-1 εάν φτάσει στο τέλος του αρχείου), άρα χρειάζεται μετατροπή σε `char`:

```
char c = (char) inputStream.read();
```

  ή απευθείας εκχώρηση του χαρακτήρα σε τύπο `int`:

```
int c = inputStream.read();
```

- Κλείσιμο αρχείου:

```
inputStream.close();
```

Παράδειγμα εισόδου από αρχείο:

Πρόγραμμα που μετράει τον αριθμό γραμμών, λέξεων και χαρακτήρων σε ένα αρχείο κειμένου.

```
import java.io.*;  // για χρήση αρχείων
import java.util.*;

public class WordCount {
{
   public static void main(String [] args)
   {
      Scanner sc = new Scanner(System.in);
      System.out.print("Δώσε όνομα αρχείου: ");
      String fileName = sc.nextLine();

      FileReader inputStream = null;

     int nc = 0, nl = 0, nw = 0; // μετρητές χαρακτήρων,γραμμών,λέξεων
      boolean insideWord = false;

      try {
           inputStream = new FileReader(fileName);
```

```
            int c;
            while ((c = inputStream.read()) != -1)
            {
                  nc++;
                  if (c == '\n')
                        nl++;
                  if (c == ' ' || c == '\n' || c == '\t')
                        insideWord = false;
                  else if (!insideWord)
                  {
                        insideWord = true;
                        nw++;
                  }
            }
            if (inputStrem != null)
                  inputStream.close();
          }
      catch (FileNotFoundException e) {
            System.out.println("Cannot find or open the file " + fileName);
      }
      System.out.println("Γραμμές: "+nl+" Λέξεις: "+nw+" Χαρακτήρες: "+nc);
   }
}
```

Παράδειγμα με είσοδο και έξοδο ανά χαρακτήρα: κωδικοποίηση ROT13

Στην κωδικοποίηση ROT13 (ROTate 13) τα γράμματα Α έως Μ (κεφαλαία ή μικρά) απεικονίζονται στα Ν έως Ζ, και το αντίστροφο. Είναι γνωστή και ως κώδικας του Καίσαρα, γιατί ο Ιούλιος Καίσαρας τη χρησιμοποιούσε για να ανταλλάσσει κρυπτογραφημένα μηνύματα.

Στο πρόγραμμα που ακολουθεί θεωρούμε ότι το όνομα του αρχείου εισόδου δίνεται στη γραμμή εντολών μέσω του ορίσματος `args` της `main`. Θεωρούμε επίσης ότι δεν χειριζόμαστε στη `main` τις εξαιρέσεις `IOException`.

```
import java.io.*;

public class ROT13
{
    public static void main(String args[]) throws IOException
    {
        if (args.length == 0) System.exit(-1);
        FileReader in = null;
        FileWriter out = null;
        try {
            in = new FileReader(args[0]);
            out = new FileWriter(args[0]+".rot13");
            int c;
            while ((c = in.read()) != -1)
               if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
                    out.write(c+13);
               else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
                    out.write(c-13);
               else
                    out.write(c);
```

```
        } finally {
              if (in != null) in.close();
              if (out != null) out.close();
        }
    }
}
```

Έξοδος σε αρχείο κειμένου (ροή γραμμών)

Χρήση της κλάσης `PrintWriter` του πακέτου `java.io`.
Άρα: `import java.io.*;`

Διαδικασία εγγραφής σε αρχείο, σε 3 βήματα:

- Δημιουργία & άνοιγμα αρχείου (στην ουσία, δημιουργία ροής εξόδου):

```
PrintWriter outputStream = new PrintWriter(new
     FileWriter("όνομα αρχείου"));
```

- Εγγραφή στο αρχείο:

```
outputStream.println("...");
```

- Κλείσιμο αρχείου:

```
outputStream.close();
```

Κάποιες παρατηρήσεις:

- Ο κατασκευαστής της `PrintWriter` δεν μπορεί να δεχθεί ως όρισμα string με το όνομα του αρχείου, για αυτό και χρησιμοποιείται ένα αντικείμενο της κλάσης `FileWriter`, της οποίας ο κατασκευαστής μπορεί να δεχθεί όνομα αρχείου.

- Το αρχείο, αν δεν υπάρχει, δημιουργείται. Αν υπάρχει ήδη, τότε δημιουργείται εκ νέου, άρα το αρχικό αρχείο διαγράφεται (χάνεται).

- Για εγγραφή δεδομένων στο τέλος υπάρχοντος αρχείου (χωρίς δηλαδή να χαθούν τα δεδομένα που έχει ήδη το αρχείο):

```
PrintWriter outputStream = new PrintWriter(new
     FileWriter("όνομα αρχείου", true));
```

  Αν ως δεύτερο όρισμα εισόδου στον κατασκευαστή της `FileOutputStream` δοθεί `boolean` μεταβλητή με τιμή `false`, τότε η ροή εξόδου λειτουργεί όπως πριν, δηλαδή σαν να μην υπήρχε δεύτερο όρισμα, άρα διαγράφεται το αρχείο σε περίπτωση που υπάρχει ήδη.

- Αφού ολοκληρωθεί η εγγραφή δεδομένων στο αρχείο, θα πρέπει πάντα να εκτελείται η εντολή που κλείνει το αρχείο, ώστε να διατηρηθεί η γενικότερη ευστάθεια του (λειτουργικού) συστήματος.

- Συνήθως, η δημιουργία της ροής της `PrintWriter` γίνεται σε δύο στάδια, με χρήση try-block ώστε να μπορεί να γίνει χειρισμός πιθανής εξαίρεσης. Δηλαδή:

εκτός try:
```
PrintWriter outputStream = null;
```
μέσα σε try:
```
try {
     outputStream = new PrintWriter(new
        FileWriter("όνομα"));
    }
catch (IOException e) {
     ...
     }
```

Η εξαίρεση IOException, θα συμβεί εάν το αρχείο για κάποιον λόγο δεν δημιουργήθηκε.

Παράδειγμα εξόδου σε αρχείο:

```
import java.io.*;  // για χρήση αρχείων
import java.util.*;    // για χρήση της Scanner
public class FileOutputDemo
{
   public static void main(String [] args)
   {
      PrintWriter outputStream = null;
      try {
           outputStream = new PrintWriter(new
                FileWriter("out.txt"));
          }
      catch (IOException e) {
         System.out.println("Error opening the file out.txt"));
         System.exit(-1);
         }
      System.out.println("Enter a line of text:");
      String line = null;
      Scanner input = new Scanner(System.in);
      line = input.nextLine();
      outputStream.println(line);
      outputStream.close();
   }
}
```

Είσοδος από αρχείο κειμένου (ροή γραμμών)

Χρήση της κλάσης `BufferedReader` του πακέτου `java.io`. Άρα: `import java.io.*;`

Διαδικασία ανάγνωσης από αρχείο, σε 3 βήματα:

- Άνοιγμα αρχείου (στην ουσία, δημιουργία ροής εισόδου):

```
BufferedReader inputStream = new BuffuredReader(new
     FileReader("όνομα αρχείου"));
```

- Ανάγνωση από το αρχείο:

```
String line = inputStream.readLine();
```

Αν φτάσει στο τέλος του αρχείου, η `readLine()` επιστρέφει την τιμή `null`.

- Κλείσιμο αρχείου:
```
inputStream.close();
```

Παράδειγμα εισόδου από αρχείο:

```
import java.io.*;  // για χρήση αρχείων

public class FileInputDemo
{
   public static void main(String [] args)
   {
      try {
           BufferedReader inputStream = new BufferedReader(new
                FileReader("data.txt"));
           String line = null;
           line = inputStream.readLine();
           System.out.println("The first line in data.txt is:");
           System.out.println(line);
           inputStream.close();
          }
      catch (FileNotFoundException e) {
           System.out.println("Cannot find of open the file data.txt!");
          }
   }
}
```

Παράδειγμα με είσοδο και έξοδο ανά γραμμή: μετατροπή κειμένου σε κεφαλαία

Στο πρόγραμμα που ακολουθεί θεωρούμε ότι το όνομα του αρχείου εισόδου δίνεται στη γραμμή εντολών μέσω του ορίσματος `args` της `main`. Θεωρούμε επίσης ότι δεν χειριζόμαστε στη `main` τις εξαιρέσεις `IOException`.

```
import java.io.*;

public class ToUpperCaseFile
{
    public static void main(String args[]) throws IOException
{
        if (args.length == 0) System.exit(-1);
        BufferedReader in = null;
        PrintWriter out = null;
        try {
            in = new BufferedReader(new FileReader(args[0]));
            out = new PrintWriter(new FileWriter(args[0].toUpperCase()));
            String line;
            while ((line = in.readLine()) != null) {
                out.println(line.toUpperCase());
            }
        } finally {
                if (in != null)
```

```
                in.close();
            if (out != null)
                out.close();
        }
    }
```

(Το πρόγραμμα αποθηκεύει το τροποποιημένο κείμενο σε αρχείο με όνομα ίδιο με το όνομα του αρχικού αρχείου αλλά με κεφαλαία γράμματα).

Άρα, συνοπτικά, οι κλάσεις και οι αντίστοιχες μέθοδοι που χρησιμοποιούνται, ανά περίπτωση:

		Ροή χαρακτήρων	**Ροή γραμμών**
ΕΙΣΟΔΟΣ	Κλάση	`FileReader`	`BufferedReader`
	Μέθοδος	`read()`	`readLine()`
ΕΞΟΔΟΣ	Κλάση	`FileWriter`	`PrintWriter`
	Μέθοδος	`write()`	`println()`

<u>Εισαγωγή ονόματος αρχείου και διαδρομής (path)</u>

Όπως φάνηκε και στο πρώτο παράδειγμα προγράμματος στην αρχή της ενότητας, το όνομα αρχείου μπορεί να εκχωρηθεί σε μια string μεταβλητή:

```
String fileName = "data.txt";
```

ή να ζητείται από τον χρήστη:

```
Scanner input = new Scanner(System.in);
System.out.println("Enter file name:");
String fileName = input.next();
```

και στη συνέχεια να χρησιμοποιείται το όνομα της string μεταβλητής που κρατάει το όνομα του αρχείου:

```
BufferedReader inputStream = new BufferedReader(new
     FileReader(fileName));
```

Επίσης, στο όνομα μπορεί να συμπεριληφθεί και η διαδρομή (path) του αρχείου στο σύστημα αρχείων του υπολογιστή. Η αναπαράσταση της διαδρομής σε διάφορα λειτουργικά συστήματα είναι ως εξής:

a) σε Unix/Linux: /user/john/home/work1/data.txt
b) σε Windows: D:\homework\hw1\data.txt

Οι αντίστοιχες εισαγωγές των διαδρομών στη Java θα ήταν αντίστοιχα ως εξής:

a)
```
... new FileReader("/home/John/Java/hw1/data.txt"));
```

b)
```
... new FileReader("D:\\Users\\John\\Java\\hw1\\data.txt"));
```
ή
```
... new FileReader("D:/Users/John/Java/hw1/data.txt"));
```

Προσοχή: Αν στα Windows το path/όνομα αρχείου *ζητηθεί από τον χρήστη*, θα πρέπει να δοθεί είτε με τον δεύτερο τρόπο (της (b) περίπτωσης παραπάνω) είτε με τον πρώτο τρόπο αλλά χρησιμοποιώντας μία κάθετο αντί για δύο, δηλαδή όπως είναι η κανονική αναπαράσταση του path στα Windows:

```
Enter file name:
D:\Users\John\Java\hw1\data.txt
```

και όχι με διπλή κάθετο.

Η κλάση `StringTokenizer`

Για την ανάγνωση μεμονωμένων λέξεων από αρχείο και όχι ολόκληρων γραμμών, χρησιμοποιούμε την κλάση `StringTokenizer` του πακέτου `java.util`.

Βασικές μέθοδοι:

`nextToken()` → επιστρέφει την επόμενη λέξη (String)
`hasMoreTokens()` → επιστρέφει `true` αν υπάρχουν αδιάβαστες λέξεις
επιστρέφει `false` αν έχουν διαβαστεί όλες οι λέξεις

Άρα, αφού διαβάσουμε μια γραμμή ενός αρχείου με την `BufferedReader`, μπορούμε να χρησιμοποιήσουμε την `StringTokenizer` στο string της γραμμής για να διαβάσουμε κάθε λέξη ξεχωριστά. Συνήθης χρήση:

```
String str = "……";
StringTokenizer st = new StringTokenizer(str);
while (st.hasMoreTokens())
   System.out.println(st.nextToken());
```

Υπάρχει η δυνατότητα να καθορίσουμε *εμείς* ποια θα είναι τα διαχωριστικά των λέξεων (το default είναι το κενό):

Αυτό γίνεται με τη χρήση του ακόλουθου κατασκευαστή της `StringTokenizer`:

```
public StringTokenizer(String str, String delimiters)
```

π.χ., `StringTokenizer st = new StringTokenizer(str, " \n.,");`

οπότε, σε αυτή την περίπτωση διαχωριστικά είναι: το κενό, η αλλαγή γραμμής, η τελεία και το κόμμα.