Αντικειμενοστραφής Προγραμματισμός

Β' εξάμηνο – Τμήμα Πληροφορικής – Ιόνιο Πανεπιστήμιο

Ενότητα 4

Μέθοδοι

Δημήτρης Ρίγγας

Μηχανικός Η/Υ & Πληροφορικής, MSc, PhD

Ε.Δι.Π. Τμήματος Πληροφορικής – Ιόνιο Παν/μιο

riggas@ionio.gr

Μέθοδοι — Εισαγωγή

Συμπεριφορά μιας κλάσης: «*τι μπορεί να κάνει;*»

Αλληλεπίδραση μεταξύ κλάσεων

Ένα αντικείμενο ζητά από ένα άλλο μια υπηρεσία αποστέλλοντας ένα **μήνυμα**.

Το μήνυμα αποτελείται από:

- **όνομα μηνύματος** = όνομα μεθόδου
- **προαιρετικό αριθμό ορισμάτων** εντός παρενθέσεων

“ Το αντικείμενο που παρέχει την υπηρεσία αποφασίζει ποια μέθοδός του θα εκτελεστεί, ταιριάζοντας το όνομα του μηνύματος με τα ορίσματά του. ”

Γιατί χωρίζουμε κώδικα σε μεθόδους;

Η διαίρεση κώδικα σε μεθόδους παρέχει σημαντικά πλεονεκτήματα:

- Μεγάλα προγράμματα → συνθέτουμε **μικρές μεθόδους**
- Κάθε μέθοδος αναπτύσσεται, ελέγχεται και **επαναχρησιμοποιείται** ανεξάρτητα
- Ο χρήστης μιας μεθόδου **δεν χρειάζεται να γνωρίζει** πώς υλοποιείται → **Αφαίρεση (Abstraction)**

Η μέθοδος `main` καλείται αυτόματα από τη JVM. Μπορούμε να ορίσουμε όσες μεθόδους θέλουμε πριν ή μετά από αυτή.

```
public class ClassMethods {
    public static void hello() { System.out.print("Hello "); }

    public static void main(String[] args) {
        hello();
        world();
    }

    public static void world() { System.out.println("world!"); }
}
```

Ορισμός μεθόδου

Γενική δομή:

```
πρόσβαση  επιστρεφόμενοςΤύπος  όνομαΜεθόδου(τύπος παράμ1, τύπος παράμ2, ...) {
    εντολές
}
```

Στοιχείο	Τιμές
Πρόσβαση	`public`, `private`, `protected`, ή χωρίς προσδιοριστή
Επιστρεφόμενος τύπος	οποιοσδήποτε πρωτογενής ή αναφερόμενος τύπος, ή `void`
Όνομα μεθόδου	οποιοδήποτε επιτρεπτό αναγνωριστικό (camelCase)

Παράδειγμα:

```
public class Calculator {
    public int add(int i1, int i2) {
        int sum = i1 + i2;
        return sum;
    }
}
```

Κλήση μεθόδου

Συνήθως μέσω ενός **αντικειμένου** της κλάσης:

```
import java.util.Scanner;

public class Calculator {
    public int add(int i1, int i2) {
        int sum = i1 + i2;
        return sum;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Provide two integers:");
        int param1 = sc.nextInt();
        int param2 = sc.nextInt();

        Calculator clt = new Calculator();
        int res = clt.add(param1, param2);  // param1→i1, param2→i2

        System.out.println(param1 + " plus " + param2 + " equals " + res);
    }
}
```

✓ Η επιστρεφόμενη τιμή κρατιέται σε **τοπική μεταβλητή**, αλλιώς χάνεται.

Στοίβα κλήσης (Call Stack)

Σειρά κλήσης: `main` → `multiply` → `add`

Εμβέλεια μεταβλητών

- Οι **τοπικές μεταβλητές** (εντός μεθόδων) χάνονται μετά την επιστροφή
- Παράμετροι = τοπικές μεταβλητές
  - **Πρωτογενείς** — αποθηκεύονται με τιμή στη στοίβα
  - **Αναφερόμενοι** — αποθηκεύουν αναφορά (δείκτη) στο σωρό

```java
public class Calculator {
    public int add(int i1, int i2) {
        int sum = i1 + i2;
        return sum;
    }
    public int multiply(int i1, int i2) {
        int multi = 0;
        for (int i = 1; i <= i1; i++)
            multi = add(multi, i2);   // <-- κλήση μεθόδου εντός μεθόδου
        return multi;
    }
    public static void main(String[] args) {
        // ...
        int res = clt.multiply(param1, param2);
    }
}
```

Εμβέλεια μεταβλητών (Scope)

Εμβέλεια = το μέρος ενός προγράμματος εντός του οποίου μια μεταβλητή είναι ορατή.

- **Τοπικές μεταβλητές** μεθόδου: εμβέλεια μέχρι το τέλος `}` της μεθόδου
- **Μεταβλητές block** (`for`, `if`, κλπ): εμβέλεια μέχρι το τέλος `}` του block

√√ Δεν επιτρέπεται δήλωση δύο μεταβλητών με το **ίδιο όνομα** με επικαλυπτόμενες εμβέλειες.

```
public class ScopeError {
    public static void main(String[] args) {
        int n = 10;
        double sum = 0.0;
        for (int i = 0; i < n; i++) {
            double n = Math.sqrt(i); // ERROR: variable n is already defined
            sum += n;
        }
    }
}
```

Πέρασμα παραμέτρων: με τιμή ή με αναφορά;

Πέρασμα με τιμή

- Αντίγραφο της τιμής περνά στη μέθοδο
- Αλλαγές **δεν** επιστρέφουν στον καλούντα

Πέρασμα με αναφορά

- Δεν δημιουργείται αντίγραφο
- Όλες οι αλλαγές επηρεάζουν το κοινό αντικείμενο
- Βελτίωση απόδοσης

Τι κάνει η Java;

Πάντα πέρασμα με τιμή, αλλά:

Τύπος	Τι περνά	Αποτέλεσμα
Πρωτογενείς (`int`, `double` …)	αντίγραφο τιμής	αλλαγές **δεν** επιστρέφουν
Αναφερόμενοι (αντικείμενα)	αντίγραφο **αναφοράς**	αλλαγές στα πεδία **επιστρέφουν**

✓ Τα ίδια τα αντικείμενα βρίσκονται στο **σωρό (heap)** και δεν μεταφέρονται.

Stack vs Heap

Stack (Στοίβα)

- Στοίβα εκτέλεσης μεθόδων
- Τοπικές μεταβλητές
- Αναφορές σε αντικείμενα
- Αυτόματη διαχείριση μνήμης (LIFO)

Heap (Σωρός)

- Όλα τα αντικείμενα δημιουργούνται εδώ (`new`)
- Διαχείριση από τον Garbage Collector

Πέρασμα παραμέτρων — Παράδειγμα

```
public class MethodParams {
    public static void swap(int a, int b) {  // δεν αλλάζει τίποτα έξω
        int temp = a; a = b; b = temp;
    }
    public static void swap(Point p) {       // αλλάζει τα πεδία του αντικειμένου
        int temp = p.x; p.x = p.y; p.y = temp;
    }
    public static void main(String[] args) {
        int x = 5, y = 3;
        System.out.printf("Πριν swap(int,int): (%d,%d)%n", x, y);
        swap(x, y);
        System.out.printf("Μετά swap(int,int): (%d,%d)%n", x, y); // ΑΜΕΤΑΒΛΗΤΟ

        Point p = new Point(5, 3);
        System.out.println("Πριν swap(Point): " + p);
        swap(p);
        System.out.println("Μετά swap(Point): " + p); // ΑΛΛΑΖΕΙ
    }
}
```

Συμπέρασμα:

- `swap(int, int)` → δεν αλλάζει `x`, `y` — αντίγραφα τιμών
- `swap(Point p)` → αλλάζει `p.x`, `p.y` — αντίγραφο αναφοράς, κοινό αντικείμενο

Static Methods

Static Class Members

Κλήση static methods — Static class members

Πώς μπορούμε να ξέρουμε πόσα αντικείμενα `Baby` έχουν δημιουργηθεί;

```java
public class Baby {
    private String name;
    private double weight = 4.0;
    private boolean isABoy;
    private int numTeeth = 0;

    // Κοινό πεδίο για ΟΛΑ τα αντικείμενα Baby στην JVM
    private static int babies = 0;

    public Baby(String name, boolean boy) {
        this.name = name;
        this.isABoy = boy;
        babies++;   // αυξάνεται κάθε φορά που δημιουργείται νέο Baby
    }

    public static int getNumberOfBabies() {
        return Baby.babies;
    }
}
```

✓ Τα `static` πεδία είναι **κοινά** για όλα τα αντικείμενα της κλάσης (class-level state).

Static methods — Κανόνες

Static μέθοδοι:

- Καλούνται **απ' ευθείας μέσω της κλάσης**: `Baby.getNumberOfBabies()`
- Έχουν πρόσβαση μόνο σε **static** πεδία/μεθόδους
- **Δεν** έχουν πρόσβαση σε non-static πεδία (δεν υπάρχει `this`)

```
public class Calculator {
    public static int add(int i1, int i2) {
        return i1 + i2;
    }

    public static void main(String[] args) {
        // Calculator clt = new Calculator(); // ΔΕΝ χρειάζεται
        int res = Calculator.add(5, 3);   // κλήση μέσω κλάσης
        System.out.println("5 + 3 = " + res);
    }
}
```

Γνωστές static κλάσεις:

- `java.lang.Math` → `Math.sqrt()`, `Math.pow()`, `Math.PI`
- `java.util.Arrays` → `Arrays.sort()`, `Arrays.toString()`

Κατασκευαστές

Constructors

Κατασκευαστές (Constructors)

Εκτελούνται κατά τη **δημιουργία** αντικειμένων με `new`.

```
public class Baby {
    String name;
    double weight = 4.0;
    boolean isABoy;
    int numTeeth = 0;

    public Baby() { }   // Default constructor – αρχικοποίηση από τιμές πεδίων
}
```

Ιδιότητες:

- Φροντίζουν για τη σωστή **αρχική διαμόρφωση** αντικειμένων
- Έχουν **ίδιο όνομα** με την κλάση
- Δεν έχουν επιστρεφόμενο τύπο
- Σχεδόν πάντα `public`

Μετά από `Baby george = new Baby();`:

- `george.name` → `null`
- `george.weight` → `4.0`
- `george.isABoy` → `false`
- `george.numTeeth` → `0`

Κατασκευαστές με παραμέτρους

```java
public class Baby {
    String name;
    double weight = 4.0;
    boolean isABoy;
    int numTeeth = 0;

    public Baby() { }  // Default constructor

    public Baby(String name, boolean boy) {
        this.name = name;       // this = το τρέχον αντικείμενο
        this.isABoy = boy;
    }
}
```

Μετά από `Baby george = new Baby("George", true);`:

- `george.name` → `"George"`
- `george.isABoy` → `true`
- `george.weight` → `4.0` (τιμή πεδίου)
- `george.numTeeth` → `0` (default)

✓ `this` διευκρινίζει ότι αναφερόμαστε στο **πεδίο** της κλάσης και όχι στην παράμετρο.

Αλυσίδα κατασκευαστών (Constructor Chaining)

Με `this(...)` ένας κατασκευαστής μπορεί να καλεί έναν άλλο.
✓ **Πρέπει να είναι η πρώτη εντολή** εντός του κατασκευαστή.

```
public class Baby {
    // ...

    public Baby(String name, boolean boy) {
        this.name = name;
        this.isABoy = boy;
    }

    public Baby(String name, boolean boy, double weight) {
        this(name, boy);          // καλεί τον παραπάνω κατασκευαστή
        this.weight = weight;
    }

    public Baby(String name, boolean boy, double weight, int numTeeth) {
        this(name, boy, weight);  // αλυσίδα κλήσεων
        this.numTeeth = numTeeth;
    }
}
```

Πλεονεκτήματα: λιγότερος κώδικας, εύκολη συντήρηση, αποφυγή επανάληψης.

Υπερφόρτωση Μεθόδων

Method Overloading

Υπερφόρτωση μεθόδων

Ορισμός: Δύο ή παραπάνω μέθοδοι/κατασκευαστές με το **ίδιο όνομα** αλλά **διαφορετική υπογραφή** (αριθμός, τύπος ή/και σειρά παραμέτρων).

√√ Δεν μπορούμε να υπερφορτώσουμε **μόνο** στον επιστρεφόμενο τύπο!

Αποδεκτό:

```
int doSomething(String s);
String doSomething(int i);
// διαφορετικοί τύποι παραμέτρων
```

Μη αποδεκτό:

```
int doSomething(String s);
String doSomething(String s);
// ίδιες παράμετροι – compile error
```

Πλεονεκτήματα:

- Οικονομία ονομάτων
- Ευανάγνωστος κώδικας
- Βιβλιοθήκες με λιγότερα ονόματα μεθόδων

Κίνδυνοι:

- Παραπλανητικά μηνύματα compiler
- Ασάφεια κατά την κλήση

Ανάλυση κλήσεων υπερφορτωμένων μεθόδων

Ο compiler αναζητά τη **βέλτιστη ταύτιση** τύπων — επιδιώκει μόνο **διεύρυνση** (widening), όχι στένωση τύπων.

```
public class ConfusedCalculator {
    public double multiply(int i1, double i2)   { return i1 * i2; }
    public double multiply(double d1, int d2)   { return d1 * d2; }

    public static void main(String[] args) {
        ConfusedCalculator clt = new ConfusedCalculator();

        clt.multiply(5, 10.0);    // ✅ → multiply(int, double)
        clt.multiply(5.0, 10);    // ✅ → multiply(double, int)
        clt.multiply(5, 10);      // ❌ Ambiguous – compile error!
        clt.multiply((double)5, 10); // ✅ explicit cast → multiply(double, int)
    }
}
```

✓ Με **explicit type casting** μπορούμε να υποδείξουμε ποια υπερφόρτωση θέλουμε.

Υπερφόρτωση toString()

Η `toString()` προέρχεται από την κλάση `Object` και επιστρέφει `ClassName@hexAddress`.

Οι `print()` / `println()` καλούν αυτόματα `toString()`.

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

    @Override                              // ← σωστή annotation
    public String toString() {
        return "(" + x + ", " + y + ")";
    }

    public static void main(String[] args) {
        System.out.println(new Point(9, 0));   // εκτυπώνει: (9, 0)
        // χωρίς override: Point@7852e922
    }
}
```

✓ **Σημείωση:** Η annotation `@Override` δεν είναι απαραίτητη αλλά **συνιστάται ανεπιφύλακτα** — ο compiler επαληθεύει ότι πράγματι κάνουμε override και όχι overload.

Υπερφόρτωση equals() — Προσοχή!

Η `equals()` της κλάσης `Object` ελέγχει **ταυτότητα αναφοράς** (ίδια θέση μνήμης).

```
class Point {
    int x, y;
    // ...

    // ΛΑΘΟΣ: υπερφόρτωση, ΟΧΙ override της Object.equals()
    public boolean equals(Point other) {
        return this.x == other.x && this.y == other.y;
    }

    // ΣΩΣΤΟ: override της Object.equals()
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof Point)) return false;
        Point other = (Point) obj;
        return this.x == other.x && this.y == other.y;
    }
}
```

√√ Χωρίς `@Override` και `Object` ως παράμετρο, δημιουργούμε **νέα υπερφόρτωση** αντί να κάνουμε override — η `Collections.contains()` και άλλες μέθοδοι δεν θα λειτουργούν σωστά!

Σύμβαση equals() & hashCode()

Όταν κάνουμε override την `equals()`, **πρέπει** να κάνουμε override και την `hashCode()`.

Σύμβαση:

- Αν `a.equals(b)` → `true`, τότε `a.hashCode() == b.hashCode()`
- Χρησιμοποιείται από `HashMap`, `HashSet`, `Hashtable`

```
@Override
public int hashCode() {
    return 31 * x + y;   // απλή υλοποίηση
    // Ή: return Objects.hash(x, y);  // προτιμώμενη σύγχρονη γραφή
}
```

√ Αν παραλείψετε το `hashCode()`, η κλάση σας δεν θα λειτουργεί σωστά σε `HashMap` / `HashSet`, ακόμα κι αν η `equals()` είναι σωστή.

Override vs Overload — Σύγκριση

Overloading (Υπερφόρτωση)

- **Ίδια κλάση**, ίδιο όνομα, **διαφορετική υπογραφή**
- Αποφασίζεται **compile-time** (static dispatch)
- ΔΕΝ χρησιμοποιεί `@Override`
- Χρήση: πολλαπλές εκδοχές της ίδιας πράξης

```
class Calculator {
    int add(int a, int b)       { return a + b; }
    double add(double a, double b) { return a + b; }
    int add(int a, int b, int c)   { return a+b+c; }
}
```

Overriding (Υπέρθεση)

- **Υποκλάση** επανορίζει μέθοδο **υπερκλάσης**
- Αποφασίζεται **runtime** (dynamic dispatch)
- Απαιτεί `@Override` + **ίδια υπογραφή**
- Χρήση: εξειδίκευση συμπεριφοράς

```
class Object {
    public String toString() { return "ClassName@hex"; }
}
class Point extends Object {
    @Override
    public String toString() { return "(" + x + "," + y + ")"; }
}
```

	Overloading	Overriding
Κλάση	ίδια	υπο/υπερκλάση
Υπογραφή	διαφορετική	**ίδια**
Επιστρεφόμενος τύπος	μπορεί να διαφέρει	ίδιος (ή covariant)
`@Override`	❌	✅ απαραίτητο
Εκτέλεση	compile-time	runtime

Μέθοδοι Μεταβλητού Πλήθους Παραμέτρων

Varargs

Varargs

Όταν το πλήθος παραμέτρων είναι άγνωστο εκ των προτέρων:

- **Με πίνακα** (verbose)
- **Με υπερφόρτωση** (επαναλαμβανόμενο)
- **Με varargs** `...` ← προτιμώμενο

```java
public class Varargs {
    public static long sumAll(int... n) {   // n αντιμετωπίζεται ως int[]
        long sum = 0;
        for (int i : n) sum += i;
        return sum;
    }

    public static void main(String[] args) {
        System.out.println(sumAll());           // 0
        System.out.println(sumAll(1));          // 1
        System.out.println(sumAll(1, 2));       // 3
        System.out.println(sumAll(1, 2, 3, 4)); // 10
    }
}
```

Κανόνες varargs:

- Μόνο **ένα** vararg ανά μέθοδο
- Πρέπει να είναι η **τελευταία** παράμετρος
- Μπορεί να συνδυαστεί με άλλες παραμέτρους:
  `void log(String prefix, int... values)`

Αναδρομικές Μέθοδοι

Recursion

Αναδρομικές μέθοδοι

Μέθοδοι που **καλούν τον εαυτό τους**.

Δομή:

1. Οργάνωση δεδομένων εισόδου
2. Επίλυση **στοιχειωδών** (base case) προβλημάτων
3. Σύνθεση για επίλυση μεγαλύτερων προβλημάτων

```java
public class Factorial {
    public static long factorial(long n) {
        if (n <= 1)                          // base case
            return 1;
        return n * factorial(n - 1);         // αναδρομική κλήση
    }

    public static void main(String[] args) {
        System.out.println("5! = " + factorial(5));   // 120
        System.out.println("10! = " + factorial(10)); // 3628800
    }
}
```

`factorial(5)` = 5 × `factorial(4)` = 5 × 4 × 3 × 2 × 1 = **120**

✓ Απαραίτητη **συνθήκη τερματισμού** — χωρίς αυτή: `StackOverflowError`!

Σύνοψη — Μέθοδοι στη Java

Σύνοψη — Μέθοδοι στη Java

Θέμα	Βασικά σημεία
Ορισμός	`πρόσβαση τύπος όνομα(παράμετροι)`
Κλήση	μέσω αντικειμένου ή (static) μέσω κλάσης
Παράμετροι	πάντα pass-by-value (αντίγραφο τιμής ή αναφοράς)
Εμβέλεια	τοπική σε μέθοδο ή block
Static	ανήκουν στην κλάση, όχι στο αντικείμενο
Κατασκευαστές	αρχικοποίηση, `this()` για chaining
Overloading	ίδιο όνομα, διαφορετική υπογραφή
`@Override`	`toString()`, `equals(Object)`, `hashCode()`
Varargs	`int... n`, πάντα τελευταία παράμετρος
Αναδρομή	base case + αναδρομική κλήση

Documentation Μεθόδων

Javadoc

Το εργαλείο javadoc

Αυτόματη παραγωγή documentation από **σχόλια ενσωματωμένα στον κώδικα**.

```java
/**
 * Υπολογίζει το άθροισμα δύο ακεραίων.
 *
 * @param i1 ο πρώτος ακέραιος
 * @param i2 ο δεύτερος ακέραιος
 * @return το άθροισμα i1 + i2
 */
public int add(int i1, int i2) {
    return i1 + i2;
}
```

Βασικά tags:

Tag	Χρήση
`@param name desc`	περιγραφή παραμέτρου
`@return desc`	περιγραφή επιστρεφόμενης τιμής
`@throws ExceptionType desc`	εξαίρεση που μπορεί να ριχτεί
`@author name`	συγγραφέας
`@since version`	από ποια έκδοση
`@see ClassName#method`	σχετική αναφορά

Δημιουργία javadoc

```
# Δημιουργία javadoc για ένα αρχείο
javadoc CashRegister.java

# Δημιουργία για ένα package
javadoc -d docs/ src/mypackage/

# Για πολλαπλά αρχεία
javadoc ../solutions/CashRegister.java
```

Το αποτέλεσμα είναι **HTML documentation** παρόμοιο με το επίσημο Java API docs.

🔗 oracle.com/technical-resources/articles/java/javadoc-tool.html

Πηγές

- Cay Horstmann, *Η γλώσσα προγραμματισμού JAVA — Αναλυτική Προσέγγιση*, Broken Hill
- Joyce Farrell, *JAVA Εκμάθηση με πρακτικά παραδείγματα*, Εκδόσεις ΚΡΙΤΙΚΗ
- Paul & Harvey Deitel, *Java How to Program*, 10/e
- E. Jones, A. Marcus, E. Wu, *Introduction to Programming in Java*, MIT OCW https://ocw.mit.edu/courses/6-092-introduction-to-programming-in-java-january-iap-2010/
- Oracle Java Documentation:
  - https://docs.oracle.com/en/java/javase/21/docs/api/
  - https://docs.oracle.com/javase/tutorial/java/
  - https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html