Αντικειμενοστραφής Προγραμματισμός

Β' εξάμηνο – Τμήμα Πληροφορικής – Ιόνιο Πανεπιστήμιο

Ενότητα 8

Χειρισμός Σφαλμάτων

Δημήτρης Ρίγγας

Μηχανικός Η/Υ & Πληροφορικής, MSc, PhD

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

riggas@ionio.gr

Χειρισμός Σφαλμάτων

Exception handling

Χειρισμός Σφαλμάτων (Exception Handling)

Τι είναι:
Αντιμετώπιση **εξαιρετικών σφαλμάτων** που προκύπτουν κατά την εκτέλεση ενός προγράμματος.

Παραδείγματα:

- Αναζήτηση εκτός ορίων πίνακα
- Χειρισμός δεδομένων διαφορετικού τύπου
- Απόπειρα κλήσης μεθόδου σε αναφορά `null`
- Άνοιγμα αρχείου που δεν υπάρχει

√ Πρόκειται για **απροσδόκητες εξαιρέσεις κατά την εκτέλεση** (runtime errors), σε αντίθεση με τα **συντακτικά σφάλματα** που ανακαλύπτονται κατά τη μεταγλώττιση.

Γιατί μας ενδιαφέρει:
Η σωστή αντιμετώπιση επιτρέπει τη δημιουργία προγραμμάτων που είναι:

- **Ανθεκτικά** σε σφάλματα
- **Κομψά** στον τερματισμό τους

Παράδειγμα – Ακέραια Διαίρεση

Ένα απλό πρόγραμμα υπολογισμού πηλίκου:

```java
import java.util.Scanner;

public class IntDivisor {

    // διαιρετέος, διαιρέτης
    public static int quotient(int numerator, int denominator) {
        return numerator / denominator;
    }

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

        System.out.print("Δώσε διαιρετέο: ");
        int n = sc.nextInt();

        System.out.print("Δώσε διαιρέτη: ");
        int d = sc.nextInt();

        System.out.println("Πηλίκο: " + quotient(n, d));
    }
}
```

Ερώτηση: Τι μπορεί να πάει λάθος;

Τι Μπορεί να Πάει Λάθος;

Δύο βασικά προβλήματα:

Πρόβλημα	Αίτιο
Διαίρεση με μηδέν	Ο χρήστης δίνει `0` ως διαιρέτη
Λάθος τύπος εισόδου	Ο χρήστης δίνει κείμενο αντί για αριθμό

Αποτέλεσμα: Διαίρεση με μηδέν

```
java IntDivisor
Δώσε διαιρετέο: 3
Δώσε διαιρέτη: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at IntDivisor.quotient(IntDivisor.java:6)
    at IntDivisor.main(IntDivisor.java:18)
```

Αποτέλεσμα: Λάθος τύπος εισόδου

```
java IntDivisor
Δώσε διαιρετέο: 5
Δώσε διαιρέτη: hello
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:909)
    ...
    at IntDivisor.main(IntDivisor.java:16)
```

✓ Stack Trace: Η λίστα δείχνει κάθε μέθοδο που κλήθηκε στη στοίβα εκτέλεσης τη στιγμή του σφάλματος.

Δημιουργία Exception

Όταν συμβαίνει σφάλμα σε μια μέθοδο:

1. Η μέθοδος **διακόπτει** την κανονική της εκτέλεση
2. **Δημιουργείται** αντικείμενο τύπου `Exception` (ή υποκλάσης) που περιέχει πληροφορίες για:
    - τον τύπο του σφάλματος
    - την κατάσταση του προγράμματος τη στιγμή του σφάλματος
3. Το αντικείμενο **«πετιέται»** (`throw`)

Αναζήτηση χειριστή:

Η JVM αναζητά κώδικα που θα **«πιάσει»** (`catch`) το exception:

- Πρώτα στην τρέχουσα μέθοδο
- Έπειτα στις μεθόδους της στοίβας εκτέλεσης, μέχρι τη `main`

Αν κανείς δεν το πιάσει:
Η JVM τυπώνει το μήνυμα σφάλματος με:

- Τον τύπο του exception
- Πιθανά μηνύματα
- Πού συνέβη το σφάλμα (stack trace)

Εντολές `try … catch`

Γιατί να χειριστούμε exceptions:

- Χωρίς χειρισμό, το πρόγραμμα μπαίνει σε αβέβαιη κατάσταση
- Σε κρίσιμες λειτουργίες, ο αντικειμενοστραφής χειρισμός είναι απαραίτητος

Δομή:

```
try {
    // Κώδικας που παρακολουθείται
    // (μπορεί να δημιουργήσει exceptions)
}
catch (ExceptionTypeA ex) {
    // Χειρισμός ExceptionTypeA
    // (ή υποκλάσεών της)
}
catch (ExceptionTypeB ex) {
    // Χειρισμός ExceptionTypeB
    // (ή υποκλάσεών της)
}
```

✓ Μετά τον χειρισμό από κάποιο `catch` block, η εκτέλεση **συνεχίζει μετά το τελευταίο catch** — δεν επιστρέφει στο `try`.

Παράδειγμα `try-catch` — `IntDivisor`

```
boolean exceptionFlag = true;
do {
    try {
        System.out.print("Δώσε διαιρετέο: ");
        int n = sc.nextInt();

        System.out.print("Δώσε διαιρέτη: ");
        int d = sc.nextInt();

        System.out.println("Πηλίκο: " + quotient(n, d));
        exceptionFlag = false;

    } catch (ArithmeticException aex) {
        System.out.println("\nArithmeticException: " + aex);
        System.out.println("Δοκιμάστε πάλι. Δεν διαιρούμε με το μηδέν.\n");

    } catch (java.util.InputMismatchException imex) {
        System.out.println("\nInputMismatchException: " + imex);
        System.out.println("Δώστε δύο ΑΚΕΡΑΙΟΥΣ!\n");
        sc.nextLine(); // Παραβλέπουμε τη λάθος είσοδο
    }
} while (exceptionFlag);
```

Τι Περιέχει κάθε Block

`try` block

- Κώδικας που **ίσως οδηγήσει σε σφάλμα**
- Κώδικας που **δεν πρέπει να εκτελεστεί** αν νωρίτερα υπάρξει σφάλμα
- Ο κώδικας **από το σημείο του σφάλματος έως το `catch`** αγνοείται
- Οι τοπικές μεταβλητές του `try` block είναι **εκτός εμβέλειας** μετά

`catch` block

- Κάθε `catch` πιάνει έναν τύπο exception **(και τις υποκλάσεις του)**
- Μετά από κάθε `try` πρέπει να ακολουθεί **τουλάχιστον ένα `catch`** (ή `finally`)
- Τα `try` / `catch` / `finally` είναι blocks — ισχύει **εμβέλεια τοπικών μεταβλητών**

⚠️ Τα `catch` blocks εξετάζονται **με τη σειρά**. Πιο ειδικοί τύποι πρέπει να έρχονται **πριν** από τους γενικούς.

Multi-Catch — Πολλαπλά Exceptions σε ένα `catch` (Java 7+)

Από Java 7, μπορούμε να **συνδυάσουμε** πολλούς τύπους exceptions σε ένα `catch` block με τον τελεστή `|`:

```
try {
    // κώδικας που μπορεί να πετάξει
    // ArithmeticException ή InputMismatchException
    doSomething();

} catch (ArithmeticException | java.util.InputMismatchException ex) {
    // Κοινός χειρισμός και για τους δύο τύπους
    System.out.println("Σφάλμα εισόδου ή υπολογισμού: " + ex.getMessage());
}
```

Κανόνες:

- Η παράμετρος `ex` είναι **implicitly final** — δεν μπορεί να της ανατεθεί νέα τιμή εντός του block
- Δεν επιτρέπεται να συνδυαστούν exceptions που έχουν **σχέση κληρονομικότητας** (π.χ. `IOException | FileNotFoundException` — error)

Πλεονέκτημα: Αποφυγή επανάληψης ίδιου κώδικα χειρισμού σε πολλά `catch` blocks.

Περίπτωση `finally`

Κώδικας που **πρέπει να εκτελεστεί ανεξάρτητα** από το αποτέλεσμα:

```
try {
    // κώδικας που μπορεί να αποτύχει
} catch (ArithmeticException aex) {
    System.out.println("ArithmeticException: " + aex);
    System.out.println("Δεν διαιρούμε με το μηδέν.\n");
} catch (java.util.InputMismatchException imex) {
    System.out.println("InputMismatchException: " + imex);
    sc.nextLine();
} finally {
    System.out.println("Το σύνολο των ακεραίων δεν είναι κλειστό " +
                       "ως προς τη διαίρεση!");
}
```

Το `finally` εκτελείται όταν:

- Το `try` τελειώσει κανονικά
- Κάποιο `catch` εκτελεστεί
- Exception προκαλεί πρόωρη έξοδο από τη μέθοδο (ακόμα κι αν δεν υπάρχει αντίστοιχο `catch`)

Το `finally` *δεν* εκτελείται αν:

- Κλήθηκε `System.exit()` εντός `try` ή `catch`

Try-With-Resources (Java 7+)

Πριν την Java 7, το κλείσιμο πόρων (αρχεία, συνδέσεις κ.λπ.) απαιτούσε επίπονο κώδικα στο `finally`:

```
// Παλιός τρόπος (πριν Java 7)
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("file.txt"));
    // χρήση br...
} finally {
    if (br != null) br.close(); // πρέπει να γίνει χειροκίνητα
}
```

Με `try-with-resources` ο πόρος κλείνει **αυτόματα**:

```
// Java 7+ – ο πόρος δηλώνεται στην παρένθεση του try
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    System.out.println("Σφάλμα ανάγνωσης: " + e.getMessage());
}
// Το br.close() καλείται αυτόματα!
```

Απαίτηση: Ο πόρος πρέπει να υλοποιεί το interface `AutoCloseable` (ή `Closeable`).
Πολλαπλοί πόροι: Διαχωρίζονται με `;` — κλείνουν με **αντίστροφη σειρά** δήλωσης.

Προειδοποίηση για Πιθανά Exceptions — `throws`

Ορθή πρακτική: Μια μέθοδος είτε:

- **Πιάνει** τα exceptions που μπορεί να δημιουργήσει, ή
- **Προειδοποιεί** τον καλούντα με `throws`

```
public static int quotient(int numerator, int denominator)
        throws ArithmeticException {
    return numerator / denominator;
}
```

Κανόνες:

- Το `throws` δηλώνεται **μετά την υπογραφή** της μεθόδου και πριν τα άγκιστρα
- Μπορεί να δηλώνει **πολλαπλά exceptions** διαχωρισμένα με κόμμα
- Για **ελεγχόμενα exceptions** (checked), η δήλωση `throws` είναι **υποχρεωτική** αν δεν τα χειριζόμαστε εντός της μεθόδου

```
// Παράδειγμα με πολλαπλά checked exceptions
public void readAndParse(String filename)
        throws IOException, NumberFormatException {
    // ...
}
```

Σκόπιμο Πέταγμα Exception — `throw`

Μπορούμε να **πετάμε exceptions** όποτε κρίνουμε ότι κάτι μη διαχειρίσιμο πρόκειται να συμβεί:

```
public static int quotient(int numerator, int denominator)
        throws ArithmeticException {

    if (denominator == 0)
        throw new ArithmeticException(\
            "Ο διαιρέτης δεν μπορεί να είναι μηδέν (/0)!");

    return numerator / denominator;
}
```

Πότε χρησιμοποιούμε `throw`:

- Όταν **εντοπίζουμε** μια μη έγκυρη κατάσταση πριν το σφάλμα συμβεί
- Εντός `catch` block που **δεν μπορεί πλήρως** να διαχειριστεί το σφάλμα (re-throw)

```
try {
    throw new RuntimeException("αρχικό πρόβλημα"); // ←
                                                   // χάνεται!
} finally {
    throw new RuntimeException("πρόβλημα στο finally"); // ←
    //                         μόνο αυτό φτάνει στον καλούντα
}
// Ο καλών βλέπει μόνο "πρόβλημα στο finally" –
// το "αρχικό πρόβλημα" εξαφανίζεται τελείως
```

Το `finally` εκτελείται πάντα, ακόμα και αν γίνει `throw` εντός `try` ή `catch`.
Η Java *επιτρέπει* συντακτικά `throw` εντός `finally`, αλλά είναι **επικίνδυνο** γιατί καταστρέφει σιωπηλά το αρχικό exception:

Το ίδιο ισχύει και για `return` εντός `finally`.

✓ Αποφεύγετε `throw` και `return` εντός `finally` block — καταπνίγουν σιωπηλά το αρχικό exception, κάνοντας το debugging εξαιρετικά δύσκολο.

Exception Chaining — Αλυσίδα Exceptions

Μπορούμε να **«τυλίξουμε»** ένα χαμηλού επιπέδου exception σε ένα υψηλότερου επιπέδου, διατηρώντας την αρχική αιτία:

```
public void loadConfiguration(String filename) throws AppConfigException {
    try {
        BufferedReader br = new BufferedReader(new FileReader(filename));
        // ανάγνωση ρυθμίσεων...
    } catch (IOException e) {
        // Τυλίγουμε το IOException σε domain-specific exception
        throw new AppConfigException("Αδυναμία φόρτωσης ρυθμίσεων: " + filename, e);
    }
}

// Αργότερα στον κώδικα:
try {
    loadConfiguration("config.xml");
} catch (AppConfigException e) {
    System.out.println(e.getMessage());
    System.out.println("Αιτία: " + e.getCause()); // το αρχικό IOException
}
```

Πλεονεκτήματα:

- Παρέχει **νόημα στο επίπεδο του domain** χωρίς να χάνει την πληροφορία debugging
- Ο καλών δεν χρειάζεται να γνωρίζει τις λεπτομέρειες υλοποίησης

Κλάσεις Throwable, Exception & Error

Throwable

`Throwable` είναι η **υπερκλάση** όλων των errors και exceptions στη Java.

Μόνο αντικείμενα `Throwable` (ή υποκλάσεών της) μπορούν:

- Να «πεταχτούν» με `throw`
- Να είναι παράμετρος σε `catch`

Δύο άμεσοι απόγονοι:

Κλάση	Χρήση	Χειρισμός
`Error`	Εξαιρετικά σπάνια σφάλματα εντός της JVM	Δεν πρέπει να «πιάνονται» — π.χ. `OutOfMemoryError`, `StackOverflowError`
`Exception`	Εξαιρετικές καταστάσεις εντός του προγράμματος	Τα προγράμματα **μπορούν και οφείλουν** να διαχειρίζονται

Ιεραρχία Κλάσεων (Μέρος)

```
Object
└── Throwable
    ├── Error  ← μη-ελεγχόμενο
    │   ├── VirtualMachineError
    │   │   ├── OutOfMemoryError
    │   │   └── StackOverflowError
    │   └── AssertionError
    │
    └── Exception
        ├── IOException  ← ελεγχόμενο
        │   ├── FileNotFoundException
        │   └── MalformedURLException
        ├── SQLException  ← ελεγχόμενο
        │
        └── RuntimeException  ← μη-ελεγχόμενο
            ├── ArithmeticException
            ├── NullPointerException
            ├── ArrayIndexOutOfBoundsException
            ├── ClassCastException
            ├── NumberFormatException
            ├── IllegalArgumentException
            ├── IllegalStateException
            └── InputMismatchException
```

Ελεγχόμενα & Μη-Ελεγχόμενα Exceptions

Η Java διακρίνει δύο κατηγορίες:

Κατηγορία	Τύποι	Υποχρέωση
Μη-ελεγχόμενα (unchecked)	`RuntimeException` και υποκλάσεις, `Error` και υποκλάσεις	Δεν απαιτείται `try-catch` ή `throws`
Ελεγχόμενα (checked)	Υποκλάσεις `Exception` που **δεν** είναι `RuntimeException`	Ο compiler **απαιτεί** `try-catch` ή `throws`

Παραδείγματα:

```
// Checked – ο compiler απαιτεί χειρισμό
try {
    FileReader fr = new FileReader("data.txt"); // IOException (checked)
} catch (IOException e) { /* ... */ }

// Unchecked – δεν απαιτεί χειρισμό (αλλά συνιστάται)
int[] arr = new int[5];
int x = arr[10]; // ArrayIndexOutOfBoundsException (unchecked)
```

✓ **Φιλοσοφία:** Ελεγχόμενα exceptions αφορούν **αναμενόμενες αποτυχίες** (αρχείο δεν βρέθηκε). Μη-ελεγχόμενα αφορούν **λάθη προγραμματισμού** (null pointer, λάθος index).

Βελτιωμένα Μηνύματα NullPointerException (Java 14/15+)

Από Java 14, τα `NullPointerException` περιέχουν **ακριβείς πληροφορίες** για το ποια αναφορά ήταν null:

```
// Πριν Java 14:
// Exception in thread "main" java.lang.NullPointerException

// Java 14+:
// Exception in thread "main" java.lang.NullPointerException:
//   Cannot invoke "String.length()" because "str" is null
```

Παράδειγμα:

```
String str = null;
int len = str.length(); // NullPointerException

// Java 14+ μήνυμα:
// Cannot invoke "String.length()" because "str" is null
```

Δημιουργία Νέου Τύπου Exception

Αξιοποιώντας κληρονομικότητα, δημιουργούμε **domain-specific** exceptions:

```
public class InsufficientFundsException extends Exception {

    private double amount; // το ποσό που λείπει

    public InsufficientFundsException() {
        super();
    }

    public InsufficientFundsException(String message) {
        super(message);
    }

    public InsufficientFundsException(String message, Throwable cause) {
        super(message, cause);
    }

    public InsufficientFundsException(Throwable cause) {
        super(cause);
    }

    // Επιπλέον πληροφορίες domain
    public InsufficientFundsException(double amount) {
        super("Ανεπαρκές υπόλοιπο: λείπουν " + amount + " €");
        this.amount = amount;
    }

    public double getAmount() { return amount; }
}
```

Ένα Exception ως Πηγή Πληροφοριών

Βασικές μέθοδοι της κλάσης `Throwable`:

Μέθοδος	Περιγραφή
`String getMessage()`	Επιστρέφει το μήνυμα λάθους του exception
`Throwable getCause()`	Επιστρέφει την αιτία (cause) ή `null`
`void printStackTrace()`	Τυπώνει το stack trace στο standard error
`String toString()`	Επιστρέφει κλάση + μήνυμα
`StackTraceElement[] getStackTrace()`	Επιστρέφει τα στοιχεία του stack trace

```java
try {
    // ...
} catch (Exception e) {
    System.err.println("Μήνυμα: " + e.getMessage());
    System.err.println("Αιτία:  " + e.getCause());
    e.printStackTrace(); // πλήρες stack trace στο stderr
}
```

✓ **Καλή πρακτική:** Σε production κώδικα χρησιμοποιούμε **logging framework** (π.χ. Log4j, SLF4J) αντί για `printStackTrace()`.

Stack Walking API (Java 9+)

Από Java 9, υπάρχει το `StackWalker` API για **προγραμματιστική ανάλυση** του stack trace:

```
// Παλιός τρόπος – δαπανηρός (δημιουργεί ολόκληρο array)
StackTraceElement[] stack = Thread.currentThread().getStackTrace();

// Java 9+ – lazy, efficient
StackWalker walker = StackWalker.getInstance();

// Βρες την κλάση που κάλεσε
Optional<String> caller = walker.walk(frames ->
    frames.skip(1)
          .findFirst()
          .map(StackWalker.StackFrame::getClassName)
);

// Λήψη του πλήρους stack
walker.walk(frames -> {
    frames.forEach(f ->
        System.out.println(f.getClassName() + "." + f.getMethodName()
                           + ":" + f.getLineNumber())
    );
    return null;
});
```

Πλεονεκτήματα: Lazy evaluation (δεν φορτώνει ολόκληρο το stack), filtering on-the-fly, καλύτερη απόδοση.

Πλεονεκτήματα Χειρισμού με Exceptions

Πλεονέκτημα 1: Διαχωρισμός Λειτουργικού Κώδικα

Χωρίς exceptions — ο κώδικας χειρισμού σφαλμάτων «πνίγει» τη λογική:

```
errorCodeType readFile {
    if (theFileIsOpen) {
        if (gotTheFileLength) {
            if (gotEnoughMemory) {
                if (readFailed) { errorCode = -1; }
            } else { errorCode = -2; }
        } else { errorCode = -3; }
        if (theFileDidntClose && errorCode == 0) { errorCode = -4; }
    } else { errorCode = -5; }
    return errorCode;
}
```

Με exceptions — η λειτουργική λογική είναι ξεκάθαρη:

```
void readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed)       { doSomething; }
      catch (sizeDeterminFailed)   { doSomething; }
      catch (memoryAllocFailed)    { doSomething; }
      catch (readFailed)           { doSomething; }
      catch (fileCloseFailed)      { doSomething; }
}
```

Πλεονέκτημα 2: Διάδοση στη Στοίβα Εκτέλεσης

Τα exceptions **«ταξιδεύουν»** αυτόματα στις προηγούμενες μεθόδους της στοίβας εκτέλεσης:

- Ο χειρισμός γίνεται **εκεί που έχει νόημα**, όχι απαραίτητα εκεί που συνέβη
- Δεν χρειάζεται κάθε μέθοδος να ελέγχει και να προωθεί κωδικούς σφαλμάτων

Πλεονέκτημα 3: Ομαδοποίηση Χειρισμού

Exceptions ίδιας ιεραρχίας μπορούν να **πιαστούν μαζί**:

```
// Και τα δύο είναι υποκλάσεις IOException:
try {
    URL url = new URL(address);          // → MalformedURLException
    FileInputStream fis = new FileInputStream("data.txt"); // → FileNotFoundException
} catch (IOException e) {
    // Κοινός χειρισμός – το αναμενόμενο αρχείο/URL δεν είναι διαθέσιμο
    System.out.println("Αδύνατη η πρόσβαση στον πόρο: " + e.getMessage());
}
```

✓ **Προσοχή στην υπερβολική γενίκευση!** Το `catch (Exception e)` πιάνει τα πάντα — ακόμα και NullPointerException και λάθη λογικής που δεν θέλουμε να αγνοούμε.

Καλές Πρακτικές Χειρισμού Exceptions

1. **Μην αγνοείς exceptions ποτέ** — ένα κενό `catch` block είναι κακή πρακτική:

```
catch (IOException e) { } // ✗ ΠΟΤΕ έτσι
```

2. **Μην πιάνεις ό,τι δε χειρίζεσαι** — αν δεν ξέρεις τι να κάνεις, άφησέ το να περάσει (`throws`)
3. **Χρησιμοποίησε specific τύπους** — πιο ειδικό exception = καλύτερος χειρισμός
4. **Δώσε χρήσιμα μηνύματα** στα custom exceptions σου
5. **Χρησιμοποίησε `try-with-resources`** Java 7+ για πόρους (αρχεία, συνδέσεις)
6. **Μην χρησιμοποιείς exceptions για έλεγχο ροής** — είναι ακριβά σε υπολογιστικούς πόρους
7. **Log πριν re-throw** — καταγραφή πλαισίου βοηθά στο debugging
8. **Checked για αναμενόμενες αποτυχίες** · **Unchecked για bugs προγράμματος**

Πηγές

- Cay Horstmann, *Η γλώσσα προγραμματισμού JAVA — Αναλυτική Προσέγγιση*, Broken Hill
- Joyce Farrell, *JAVA: Εκμάθηση με πρακτικά παραδείγματα*, Εκδόσεις ΚΡΙΤΙΚΗ
- Paul Deitel & Harvey Deitel, *Java How to Program*, 10th ed.
- Γρηγόρης Τσουμάκας, *Αντικειμενοστρεφής Προγραμματισμός — Ενότητα 11 — Χειρισμός σφαλμάτων*
- Oracle Java Tutorial — Exceptions
- JEP 358 — Helpful NullPointerExceptions (Java 14)
- JEP 259 — Stack-Walking API (Java 9)
- JEP 334 — JVM Constants API (Java 12)