



☕ CHƯƠNG 5

Xử Lý Ngoại Lệ (Exceptions) & Kiểm Thử (Testing)

Từ Zero đến Hero: Làm chủ Exception Handling và Unit Testing

Chào mừng bạn đến với chương học quan trọng nhất trong hành trình trở thành Java Developer chuyên nghiệp! Exception handling và Testing không chỉ là kỹ năng "nice to have" mà là **BẮT BUỘC** trong mọi dự án thực tế. Hãy sẵn sàng nâng cấp kỹ năng của bạn!

Mục Tiêu Học Tập

Sau khi hoàn thành chương này, bạn sẽ tự tin làm chủ các kỹ năng then chốt của một lập trình viên chuyên nghiệp:



Phân biệt Exceptions

Hiểu rõ sự khác biệt giữa Checked và Unchecked Exceptions, biết khi nào nên sử dụng loại nào



Xử lý lỗi đúng cách

Áp dụng try-catch-finally và try-with-resources một cách thành thạo



Tạo Custom Exception

Thiết kế exception riêng cho business logic của ứng dụng



Viết Unit Test

Sử dụng JUnit 5 để kiểm thử code một cách chuyên nghiệp



Testing Best Practices

Áp dụng các kỹ thuật testing hiện đại như Boundary Testing và Equivalence Partitioning



File I/O & Serialization

Thao tác với file và lưu trữ object một cách an toàn

Kiến Thức Cần Có

Trước khi bắt đầu hành trình này, hãy đảm bảo bạn đã nắm vững những kiến thức nền tảng sau. Đừng lo nếu còn mơ hồ - chúng ta sẽ ôn lại ngắn gọn!

Class và Object (Chương 2)

- Tạo class và định nghĩa methods
- Sử dụng access modifiers (public, private, protected)
- Hiểu về constructor và encapsulation

Inheritance (Chương 3.1)

- Kế thừa class với từ khóa extends
- Sử dụng super để gọi constructor của lớp cha
- Tạo custom class với hierarchy rõ ràng

 **Lưu ý quan trọng:** Exception handling và Testing là **kỹ năng bắt buộc** trong lập trình chuyên nghiệp. Không có công ty nào tuyển developer không biết xử lý lỗi và viết test. Hãy học kỹ phần này!

5.1. Phân Loại Ngoại Lệ

KHÁI NIỆM CƠ BẢN

Exception là gì?

Định nghĩa

Exception (Ngoại lệ) là sự kiện xảy ra trong quá trình thực thi chương trình làm gián đoạn luồng xử lý bình thường. Đây không phải là lỗi cú pháp (syntax error) mà là lỗi logic hoặc tình huống bất thường xảy ra lúc runtime.

Các tình huống phổ biến

- **ArithmetricException:** Chia một số cho 0
- **FileNotFoundException:** Truy cập file không tồn tại
- **SQLException:** Kết nối database thất bại
- **ArrayIndexOutOfBoundsException:** Truy cập mảng vượt quá chỉ số
- **NullPointerException:** Gọi method trên object null



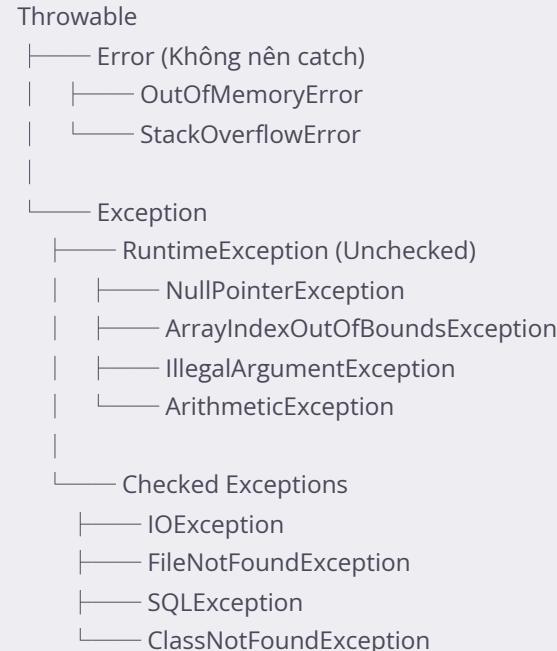
💡 Ví dụ thực tế: Máy báo cháy

Exception giống như việc máy báo cháy hú còi khi phát hiện khói:

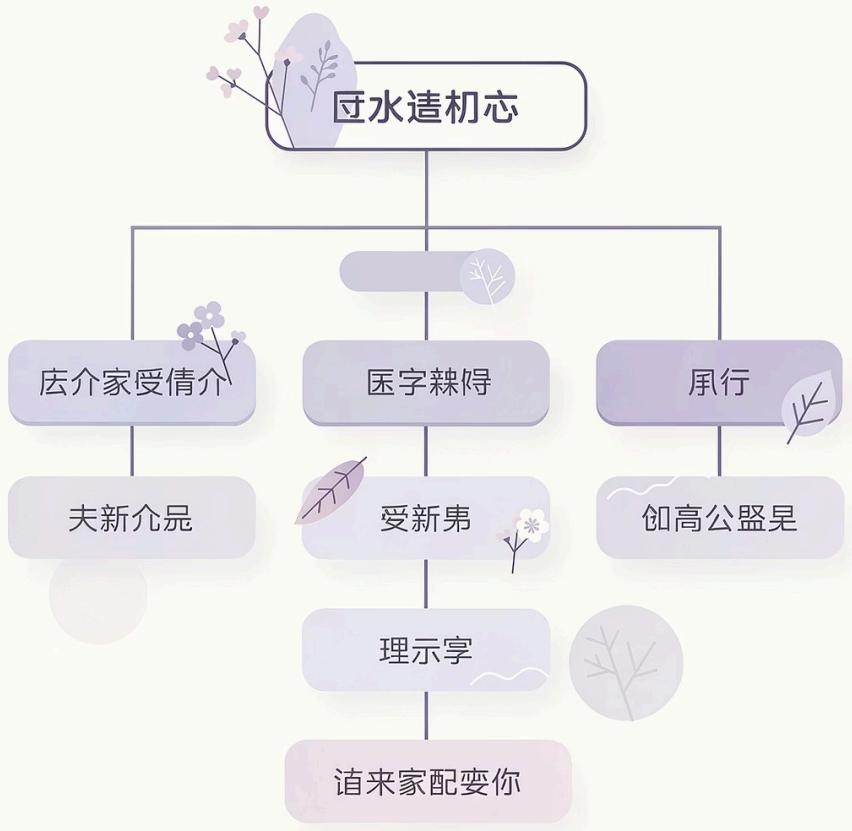
- Nó **KHÔNG** dập lửa (không tự sửa lỗi)
- Nó chỉ **BÁO ĐỘNG** để bạn biết có vấn đề
- Bạn phải **XỬ LÝ** (chạy thoát thân hoặc lấy bình cứu hỏa)
- Nếu bạn **LỜ ĐI** (không catch), ngôi nhà (chương trình) sẽ cháy rụi (crash)

Exception Hierarchy trong Java

Java tổ chức các Exception theo cấu trúc phân cấp rõ ràng. Hiểu được hierarchy này giúp bạn biết nên catch loại Exception nào và khi nào:



- ⚠ **Lưu ý: Error** là những lỗi nghiêm trọng cấp hệ thống (như hết bộ nhớ, stack overflow) mà bạn **KHÔNG NÊN** cố gắng catch. Hãy để JVM xử lý chúng. Exception mới là cái bạn cần quan tâm!



Checked Exceptions

BẮT BUỘC XỬ LÝ

Đặc điểm

- Phải được xử lý (catch) hoặc khai báo (throws)
- Compiler bắt buộc kiểm tra tại thời điểm compile
- Thường là lỗi **ngoài tầm kiểm soát** của programmer
- Ví dụ: file không tồn tại, network bị ngắt, database unavailable

Các Checked Exceptions phổ biến

- **IOException:** Lỗi đọc/ghi file hoặc network
- **FileNotFoundException:** File không tìm thấy
- **SQLException:** Lỗi database
- **ClassNotFoundException:** Class không tồn tại

Ví dụ Code

```
//
```

Unchecked Exceptions (RuntimeException)

KHÔNG BẮT BUỘC

Đặc điểm

- **Không bắt buộc** phải catch hoặc throws
- **Compiler không kiểm tra**
- Thường là **lỗi logic** của programmer
- Có thể tránh được bằng cách code cẩn thận

Ví dụ Code

```
//
```

Các Unchecked Exceptions phổ biến

- **NullPointerException:** Gọi method trên object null
- **ArrayIndexOutOfBoundsException:** Index mảng vượt quá
- **IllegalArgumentException:** Tham số không hợp lệ
- **ArithmetricException:** Chia cho 0

Ví dụ minh họa: Cảnh sát giao thông

Để hiểu rõ hơn sự khác biệt giữa Checked và Unchecked Exceptions, hãy tưởng tượng bạn đang lái xe trên đường:

Unchecked (RuntimeException): Vượt đèn đỏ

Giống như việc bạn **vượt đèn đỏ**. Đây là **lỗi của bạn** (lỗi logic trong code). Luật không bắt bạn phải đội mũ sắt để đề phòng vượt đèn đỏ, nhưng nếu vi phạm thì bị phạt (Exception xảy ra). Bạn hoàn toàn có thể tránh bằng cách chú ý quan sát.

→ Tương tự: *NullPointerException* xảy ra vì bạn quên check null trước khi dùng object.

Checked Exception: Biển "Đường đang thi công"

Giống như gặp **biển báo "Đường đang thi công"**. Đây **không phải lỗi của bạn**, nhưng luật **bắt buộc** bạn phải chạy đường vòng (catch/throws). Bạn phải chuẩn bị tinh thần xử lý tình huống này trước khi ra đường, dù không muốn.

→ Tương tự: *FileNotFoundException* - file không tồn tại không phải lỗi code của bạn, nhưng bạn phải xử lý.

So sánh Checked vs Unchecked

Bảng so sánh chi tiết giúp bạn phân biệt rõ ràng hai loại Exception:

Đặc điểm	Checked Exception	Unchecked Exception
Phải catch/throws?	✓ Có (bắt buộc)	✗ Không (tùy chọn)
Compiler kiểm tra?	✓ Có (compile-time)	✗ Không
Kế thừa từ	Exception	RuntimeException
Khi nào xảy ra	Runtime (có thể dự đoán)	Runtime (lỗi logic)
Nguyên nhân	Ngoài tầm kiểm soát	Lỗi lập trình
Có thể tránh?	✗ Khó (phụ thuộc môi trường)	✓ Có (bằng validation)
Ví dụ	FileNotFoundException, SQLException	NullPointerException, ArithmeticException

Khi nào nên BẮT lỗi (catch)?

Quyết định có nên catch một exception hay không là một nghệ thuật. Dưới đây là hướng dẫn cụ thể:



Có thể xử lý và tiếp tục

Khi bạn có phương án thay thế hoặc fallback để chương trình tiếp tục chạy bình thường sau khi gặp lỗi.



Cần log lỗi để debug

Khi bạn muốn ghi lại thông tin lỗi vào log file để phân tích sau này, nhưng vẫn muốn chương trình chạy tiếp.



Cần thông báo cho user

Khi bạn muốn hiển thị thông báo lỗi thân thiện với người dùng thay vì để chương trình crash.

Ví dụ thực tế

```
public void readConfigFile() {  
    try {  
        FileReader file = new FileReader("config.txt");  
        // Đọc file config  
    } catch (FileNotFoundException e) {  
        //
```

Khi nào nên NÉM lỗi (throw/throws)?

Không phải lúc nào cũng nên catch exception tại chỗ. Đôi khi việc ném lên cho caller xử lý là lựa chọn đúng đắn hơn:



Không thể xử lý tại chỗ

Method hiện tại không có đủ context hoặc quyền hạn để xử lý lỗi một cách hợp lý.



Muốn để caller xử lý

Caller biết rõ hơn về ngữ cảnh và có thể đưa ra quyết định xử lý phù hợp hơn.



Lỗi quan trọng không thể bỏ qua

Đây là lỗi nghiêm trọng ảnh hưởng đến toàn bộ luồng xử lý và phải được xử lý ở cấp cao hơn.

Ví dụ thực tế

```
// Method không thể xử lý lỗi
public void connectToDatabase() throws SQLException {
    // Kết nối database
    // Nếu lỗi → Ném lên cho caller xử lý
    Connection conn = DriverManager.getConnection(url);
}

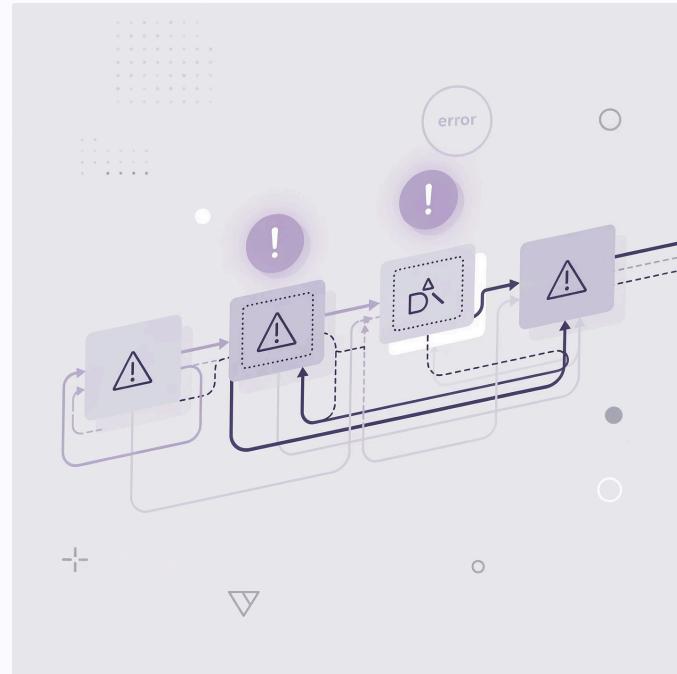
// Caller xử lý
public void initialize() {
    try {
        connectToDatabase();
    } catch (SQLException e) {
        //
    }
}
```

Khi nào nên TỰ ĐỊNH NGHĨA Exception?

Đôi khi các Exception có sẵn trong Java không đủ rõ ràng cho business logic của bạn. Đó là lúc bạn cần tạo Custom Exception:

Tự định nghĩa khi:

- Cần exception **riêng cho business logic** (ví dụ: InsufficientBalanceException)
- Muốn **thông điệp lỗi rõ ràng** hơn thay vì dùng Exception chung chung
- Cần **thêm thông tin** vào exception (ví dụ: số dư hiện tại, số tiền cần rút)
- Muốn caller có thể **catch cụ thể** để xử lý khác nhau



Ví dụ Custom Exception

```
// Tạo Custom Exception
public class InsufficientBalanceException extends Exception {
    private double balance;
    private double amount;

    public InsufficientBalanceException(double balance, double amount) {
        super("Insufficient balance. Current: " + balance + ", Required: " + amount);
        this.balance = balance;
        this.amount = amount;
    }

    public double getBalance() { return balance; }
    public double getAmount() { return amount; }
}

// Sử dụng
public void withdraw(double amount) throws InsufficientBalanceException {
    if (amount > balance) {
        throw new InsufficientBalanceException(balance, amount);
    }
    balance -= amount;
}
```

5.2. Cơ Chế Xử Lý Exception

TRY-CATCH-FINALLY

Cú pháp cơ bản

Try-catch-finally là cơ chế nền tảng để xử lý exception trong Java. Mỗi block có vai trò riêng:

01

try block

Chứa code có thể ném exception. Java sẽ "thử" chạy code này.

02

catch block

Xử lý exception nếu xảy ra. Có thể có nhiều catch block.

03

finally block

Code luôn chạy, dù có exception hay không. Dùng để cleanup (đóng file, connection...).

```
try {  
    // Code có thể ném exception  
} catch (ExceptionType e) {  
    // Xử lý exception  
} finally {  
    // Code luôn chạy (dù có exception hay không)  
}
```

Ví dụ Try-Catch-Finally chi tiết

```
public void divide(int a, int b) {  
    try {  
        int result = a / b;  
        System.out.println("Result: " + result);  
    } catch (ArithmaticException e) {  
        System.out.println("Error: Cannot divide by zero!");  
        e.printStackTrace();  
    } finally {  
        System.out.println("Division operation completed");  
    }  
}  
  
// Test  
divide(10, 2); // Result: 5, Division operation completed  
divide(10, 0); // Error: Cannot divide by zero!, Division operation completed
```

Phân tích Luồng nhảy (Jump Flow)

Khi divide(10, 0) chạy, JVM thực hiện "cú nhảy ngoạn mục":

- 
- Bước 1: Lỗi xảy ra
int result = 10 / 0; → 🔥 BÙM! ArithmaticException.
 - Bước 2: Dừng ngay
Dòng println("Result: ...") KHÔNG BAO GIỜ được chạy.
 - Bước 3: Nhảy câu
Java tìm ngay block catch phù hợp với ArithmaticException.
 - Bước 4: Đáp đất
Chạy code trong catch block. In ra "Error...".
 - Bước 5: Chốt hạ
Finally LUÔN LUÔN chạy để dọn dẹp hiện trường.

 **Bài học:** Hãy coi try-catch như một "gõ bom". try là lúc cắt dây bom. Nếu nổ (Exception), bạn bị văng ra xa và rơi vào lưới đỡ (catch), chứ không chết (Crash App).

Multiple Catch Blocks

Một try block có thể có nhiều catch block để xử lý các loại exception khác nhau. Điều quan trọng là phải sắp xếp từ **cụ thể đến chung chung**:

```
try {  
    int[] arr = new int[5];  
    int value = arr[10]; // ArrayIndexOutOfBoundsException  
  
    String str = null;  
    int len = str.length(); // NullPointerException  
  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("Array index out of bounds!");  
  
} catch (NullPointerException e) {  
    System.out.println("Null pointer exception!");  
  
} catch (Exception e) {  
    // Catch-all cho các exception khác  
    System.out.println("Other exception: " + e.getMessage());  
}
```

⚠ Lưu ý quan trọng:

- Catch từ **cụ thể → chung chung** (Exception phải ở cuối cùng)
- Nếu đặt Exception ở đầu, các catch sau sẽ không bao giờ chạy (unreachable code)
- Compiler sẽ báo lỗi nếu bạn sắp xếp sai thứ tự

Multi-Catch (Java 7+)

Từ Java 7 trở đi, bạn có thể catch nhiều exception trong một block duy nhất bằng cách sử dụng toán tử | (pipe). Điều này giúp giảm code trùng lặp khi xử lý các exception tương tự nhau:

✗ Cách cũ (trùng lặp)

```
try {  
    // Code  
} catch (FileNotFoundException e) {  
    System.out.println("File error: "  
        + e.getMessage());  
    logger.error(e);  
} catch (IOException e) {  
    System.out.println("File error: "  
        + e.getMessage());  
    logger.error(e);  
}
```

✓ Cách mới (gọn gàng)

```
try {  
    // Code  
} catch (FileNotFoundException | IOException e) {  
    System.out.println("File error: "  
        + e.getMessage());  
    logger.error(e);  
}
```

- ☐ **Lưu ý:** Các exception trong multi-catch **KHÔNG ĐƯỢC** có quan hệ kế thừa với nhau. Ví dụ: không thể multi-catch (Exception | IOException) vì IOException kế thừa từ Exception.

Try-with-Resources

JAVA 7+

Vấn đề với Finally

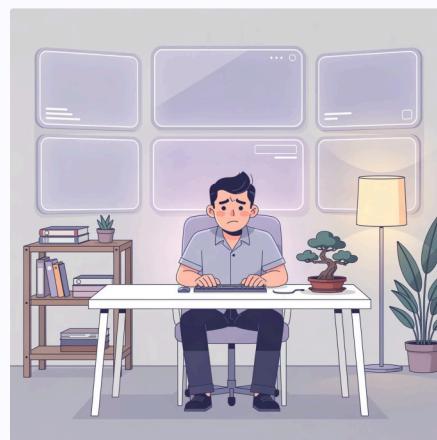
Trước Java 7, việc đóng resources (file, connection...) trong finally block rất phức tạp và dễ sai sót:

✗ Code cũ (phức tạp)

```
FileReader file = null;  
try {  
    file = new FileReader("file.txt");  
    // Đọc file  
} catch (IOException e) {  
    e.printStackTrace();  
} finally {  
    if (file != null) {  
        try {  
            file.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Vấn đề

- ✗ Code dài, phức tạp
- ✗ Nested try-catch khó đọc
- ✗ Dễ quên close resources
- ✗ Phải check null
- ✗ Có thể bị exception trong finally



Giải pháp: Try-with-Resources

Java 7 giới thiệu Try-with-Resources - một tính năng tuyệt vời giúp tự động đóng resources mà không cần finally:

Cú pháp

```
try (ResourceType resource = new ResourceType()) {  
    // Sử dụng resource  
} catch (Exception e) {  
    // Xử lý exception  
}  
// Resource tự động được close - KHÔNG CẦN finally!
```

✓ Ví dụ thực tế

```
// Đơn giản và rõ ràng  
try (FileReader file =  
    new FileReader("file.txt")) {  
    // Đọc file  
    // File tự động close  
    // khi ra khỏi try block  
} catch (IOException e) {  
    e.printStackTrace();  
}  
// Không cần finally!
```

█ Ví dụ minh họa

Try-with-resources giống như **cửa tự đóng có lò xo**:

- Bạn đi vào (mở file), làm gì thì làm
- Khi bước ra (kết thúc block), cửa tự động đóng "RÂM" lại
- Bạn **KHÔNG CẦN** quay lại đóng cửa (gọi close())
- An toàn tuyệt đối!

AutoCloseable Interface

Để có thể dùng với try-with-resources, resource phải implement interface **AutoCloseable**:

Interface định nghĩa

```
public interface AutoCloseable {  
    void close() throws Exception;  
}
```

Mọi class implement interface này đều có thể dùng trong try-with-resources.

Các class đã implement AutoCloseable

- **File I/O:** FileReader, FileWriter, BufferedReader, BufferedWriter
- **Network:** Socket, ServerSocket
- **Database:** Connection, Statement, ResultSet
- **Streams:** InputStream, OutputStream, Scanner
- Và hầu hết các class làm việc với resources khác

Multiple Resources

Bạn có thể khai báo nhiều resources cùng lúc (cách nhau bằng dấu chấm phẩy). Chúng sẽ tự động close theo **thứ tự ngược lại** với lúc khai báo:

```
try (  
    FileReader reader = new FileReader("input.txt");  
    FileWriter writer = new FileWriter("output.txt")  
) {  
    // Sử dụng reader và writer  
    // Close tự động: writer.close() → reader.close()  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Best Practices: Không bao giờ "nuốt" lỗi

CẢNH BÁO QUAN TRỌNG

Vấn đề: Swallowing Exceptions

Một trong những sai lầm **NGUY HIỂM NHẤT** mà developer mới hay mắc phải là "nuốt" exception - tức là catch nhưng không làm gì cả:

```
try {  
    processData();  
} catch (Exception e) {  
    //
```

✓ Xử lý exception đúng cách

Có 4 cách xử lý exception đúng đắn, tùy thuộc vào ngữ cảnh:

1

Log exception

```
try {  
    processData();  
} catch (Exception e) {  
    logger.error("Error processing data", e);  
    // Hoặc e.printStackTrace();  
}
```

Ghi lại thông tin lỗi để debug sau này.

2

Thông báo cho user

```
try {  
    processData();  
} catch (Exception e) {  
    System.out.println("An error occurred. " +  
        "Please try again.");  
    logger.error("Error", e);  
}
```

Hiển thị thông báo thân thiện cho người dùng.

3

Ném lại nếu không thể xử lý

```
try {  
    processData();  
} catch (Exception e) {  
    logger.error("Error", e);  
    throw new RuntimeException(  
        "Failed to process data", e);  
}
```

Để caller cấp cao hơn xử lý.

4

Wrap exception

```
try {  
    processData();  
} catch (IOException e) {  
    throw new DataProcessingException(  
        "Failed to process data", e);  
}
```

Chuyển đổi thành custom exception rõ nghĩa hơn.

5.3. Unit Testing - Tư Duy Kiểm Thủ Hiện Đại

TESTING MINDSET

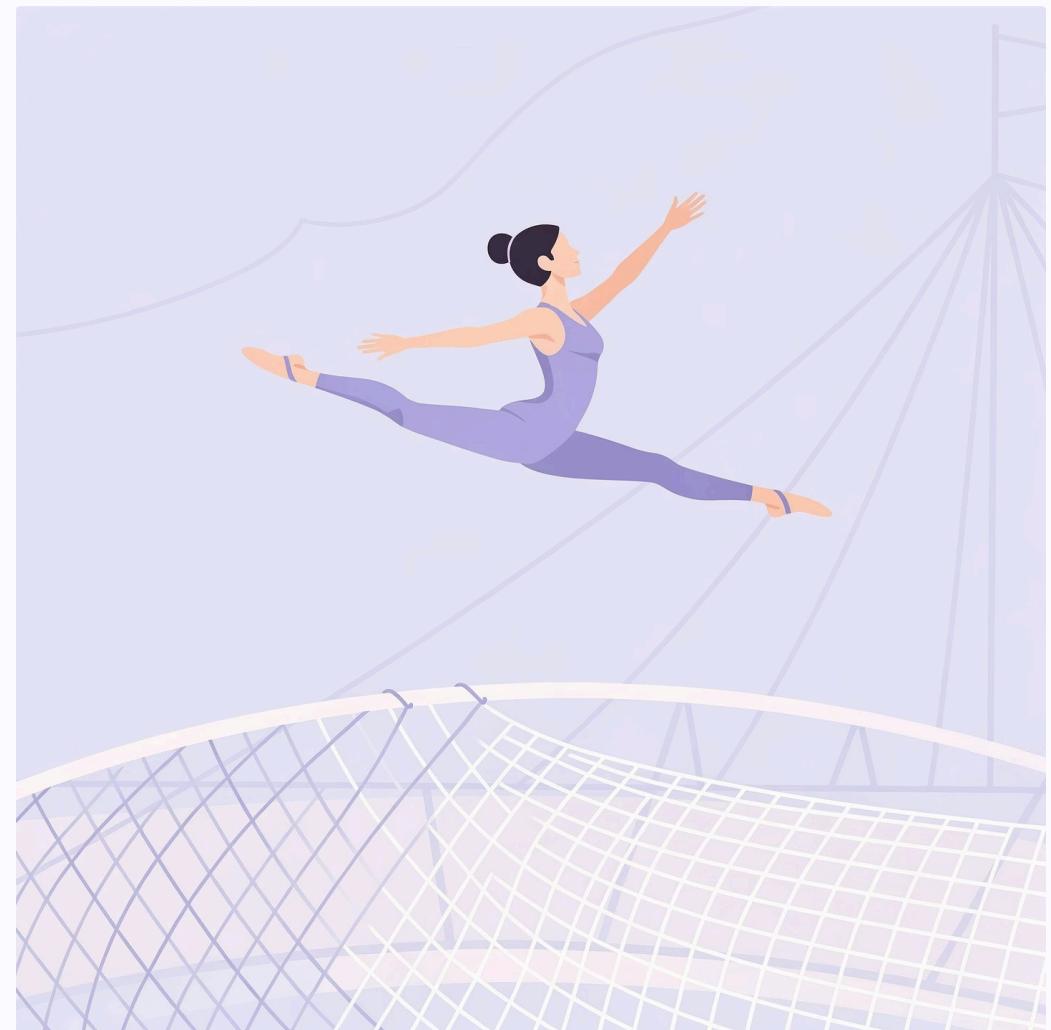
Unit Testing là gì?

Định nghĩa

Unit Test là kiểm thử từng đơn vị code (method, class) một cách độc lập để đảm bảo nó hoạt động đúng như mong đợi.

Đặc điểm

- Test một đơn vị **nhỏ nhất** (method, class)
- Chạy **nhanh** (thường < 1 giây)
- **Độc lập** (không phụ thuộc test khác)
- Có thể chạy **tự động** (CI/CD)
- **Repeatable** (chạy nhiều lần cho kết quả giống nhau)



💡 Ví dụ: Lưới an toàn

Viết code không có Unit Test giống như diễn viên xiếc đi trên dây mà **không có lưới bảo vệ**. Sơ sẩy là "chết" (bug production).

Khi có Unit Test, bạn tự tin nhảy múa, nhào lộn (Refactor code). Nếu lỡ trượt chân, tấm lưới (Test fail) sẽ đỡ bạn ngay.

Tại sao cần Unit Test?



Phát hiện lỗi sớm

Tìm bug ngay khi viết code, thay vì đợi đến khi deploy production. Chi phí sửa bug tăng theo cấp số nhân theo thời gian!



Tự tin refactor

Cải thiện code mà không sợ phá vỡ chức năng cũ. Test sẽ báo ngay nếu có gì sai.



Tài liệu sống

Test code chính là tài liệu tốt nhất giải thích code làm gì và cách dùng như thế nào.



Tăng chất lượng code

Khi viết test, bạn buộc phải suy nghĩ về design, giúp code clean và maintainable hơn.

Ví dụ đơn giản

```
// Code cần test
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}

// Unit Test
@Test
public void testAdd() {
    Calculator calc = new Calculator();
    int result = calc.add(2, 3);
    assertEquals(5, result); //
```

JUnit 5 - Framework Testing

INDUSTRY STANDARD

Giới thiệu JUnit 5

JUnit 5 là framework testing phổ biến và mạnh mẽ nhất cho Java. Nó là công cụ **BẮT BUỘC** mà mọi Java developer phải biết.

Cài đặt với Maven

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Cài đặt với Gradle

```
dependencies {
  testImplementation(
    'org.junit.jupiter:junit-jupiter:5.10.0'
  )
}

test {
  useJUnitPlatform()
}
```

- ☐ **Lưu ý:** JUnit 5 khác hoàn toàn với JUnit 4. Hãy chắc chắn bạn đang dùng JUnit 5 (`org.junit.jupiter`) chứ không phải JUnit 4 (`org.junit`).

Cấu trúc Test cơ bản - AAA Pattern

Mọi unit test đều nên tuân theo pattern **AAA** (Arrange-Act-Assert) để dễ đọc và bảo trì:

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*;  
  
public class CalculatorTest {  
  
    @Test  
    public void testAdd() {  
        // 1. Arrange (Chuẩn bị)  
        Calculator calc = new Calculator();  
  
        // 2. Act (Thực hiện)  
        int result = calc.add(2, 3);  
  
        // 3. Assert (Kiểm tra)  
        assertEquals(5, result);  
    }  
}
```



Arrange (Chuẩn bị)

Thiết lập bối cảnh. Tạo objects, chuẩn bị dữ liệu đầu vào. Giống như bạn bày biện nguyên liệu trước khi nấu ăn.



Act (Hành động)

Gọi **ĐÚNG 1** method cần test. Nếu lỗi xảy ra, bạn biết ngay do method này gây ra.



Assert (Khẳng định)

So sánh kết quả thực tế với kỳ vọng. Đây là "cam kết bảo hành" - nếu sai, code có vấn đề.

- ⚠ **Sai lầm thường gặp:** Một số bạn viết `System.out.println()` trong test để nhìn bằng mắt. **ĐÚNG** làm thế! Hãy để Assert tự động phán xét Đúng/Sai.

Các Assertions phổ biến

JUnit 5 cung cấp nhiều assertion methods để kiểm tra các điều kiện khác nhau. Dưới đây là những cái bạn sẽ dùng thường xuyên nhất:

```
import static org.junit.jupiter.api.Assertions.*;  
  
@Test  
public void testAssertions() {  
    // assertEquals - So sánh bằng  
    assertEquals(5, 2 + 3);  
    assertEquals("Hello", "Hello");  
  
    // assertNotEquals - So sánh khác  
    assertNotEquals(5, 2 + 2);  
  
    // assertTrue - Kiểm tra true  
    assertTrue(5 > 3);  
  
    // assertFalse - Kiểm tra false  
    assertFalse(5 < 3);  
  
    // assertNull - Kiểm tra null  
    String str = null;  
    assertNull(str);  
  
    // assertNotNull - Kiểm tra không null  
    String str2 = "Hello";  
    assertNotNull(str2);  
  
    // assertSame - Cùng object (same reference)  
    Object obj1 = new Object();  
    Object obj2 = obj1;  
    assertSame(obj1, obj2);  
  
    // assertNotSame - Khác object  
    Object obj3 = new Object();  
    assertNotSame(obj1, obj3);  
  
    // assertThrows - Kiểm tra exception  
    assertThrows(ArithmeticException.class, () -> {  
        int result = 10 / 0;  
    });  
}
```

Kỹ thuật viết Test Case

PROFESSIONAL TECHNIQUES

Test Case tốt có những đặc điểm gì?



Tên test mô tả rõ ràng

Đặt tên theo format: test[Method]_[Scenario]_[ExpectedResult]. Ví dụ: testAdd_PositiveNumbers_ReturnsSum



Test một điều duy nhất

Mỗi test chỉ nên kiểm tra một behavior cụ thể. Nếu test fail, bạn biết ngay vấn đề ở đâu.



Độc lập

Test không phụ thuộc vào test khác. Có thể chạy theo bất kỳ thứ tự nào.



Repeatable

Chạy nhiều lần cho kết quả giống nhau. Không phụ thuộc vào thời gian, random, external resources.



Ví dụ Test Case chuyên nghiệp

Dưới đây là ví dụ về cách viết test case đúng chuẩn, test nhiều scenarios khác nhau:

```
@Test
public void testAdd_PositiveNumbers_ReturnsSum() {
    // Arrange
    Calculator calc = new Calculator();

    // Act
    int result = calc.add(5, 3);

    // Assert
    assertEquals(8, result);
}

@Test
public void testAdd_NegativeNumbers_ReturnsSum() {
    Calculator calc = new Calculator();
    int result = calc.add(-5, -3);
    assertEquals(-8, result);
}

@Test
public void testAdd_MixedNumbers_ReturnsSum() {
    Calculator calc = new Calculator();
    int result = calc.add(5, -3);
    assertEquals(2, result);
}

@Test
public void testAdd_WithZero_ReturnsOtherNumber() {
    Calculator calc = new Calculator();
    int result = calc.add(5, 0);
    assertEquals(5, result);
}

@Test
public void testAdd_BothZero_ReturnsZero() {
    Calculator calc = new Calculator();
    int result = calc.add(0, 0);
    assertEquals(0, result);
}
```

☐💡 **Best Practice:** Viết test cho **TẤT CẢ** các trường hợp: positive, negative, zero, boundary values, null, empty... Đừng chỉ test "happy path"!

Kiểm thử Biên (Boundary Testing)

Boundary Testing là kỹ thuật test các **giá trị biên** - nơi bugs thường ẩn nấp nhất. Bạn cần test:

- Giá trị **đúng tại biên** (ví dụ: 0, 100)
- Giá trị **ngay dưới biên** (ví dụ: -1, 99)
- Giá trị **ngay trên biên** (ví dụ: 1, 101)
- Giá trị **giữa khoảng** (ví dụ: 50)

Ví dụ: Tính giảm giá theo tổng tiền

```
// Business Logic:  
// 0-99: Không giảm giá (0%)  
// 100-499: Giảm 10%  
// 500-999: Giảm 20%  
// >= 1000: Giảm 50%  
  
@Test  
public void testCalculateDiscount_BoundaryValues() {  
    DiscountCalculator calc = new DiscountCalculator();  
  
    // Biên dưới của mỗi khoảng  
    assertEquals(0, calc.calculateDiscount(0)); // 0%  
    assertEquals(10, calc.calculateDiscount(100)); // 10%  
    assertEquals(20, calc.calculateDiscount(500)); // 20%  
    assertEquals(50, calc.calculateDiscount(1000)); // 50%  
  
    // Ngay dưới biên  
    assertEquals(0, calc.calculateDiscount(99)); // 0%  
    assertEquals(10, calc.calculateDiscount(499)); // 10%  
    assertEquals(20, calc.calculateDiscount(999)); // 20%  
  
    // Ngay trên biên  
    assertEquals(10, calc.calculateDiscount(101)); // 10%  
    assertEquals(20, calc.calculateDiscount(501)); // 20%  
    assertEquals(50, calc.calculateDiscount(1001)); // 50%  
  
    // Giữa khoảng  
    assertEquals(10, calc.calculateDiscount(300)); // 10%  
    assertEquals(20, calc.calculateDiscount(750)); // 20%  
}
```

Kiểm thử Lớp tương đương (Equivalence Partitioning)

Equivalence Partitioning là kỹ thuật chia input thành các **lớp tương đương** - các giá trị trong cùng một lớp sẽ được xử lý giống nhau. Bạn chỉ cần test **một vài** giá trị đại diện cho mỗi lớp:

Ví dụ: Tính điểm chữ từ điểm số

```
// Business Logic: calculateGrade(double score)
// Lớp 1: 0-49 → "F"
// Lớp 2: 50-59 → "D"
// Lớp 3: 60-69 → "C"
// Lớp 4: 70-79 → "B"
// Lớp 5: 80-100 → "A"

@Test
public void testCalculateGrade_ClassF() {
    GradeCalculator calc = new GradeCalculator();
    // Test vài giá trị đại diện cho lớp F
    assertEquals("F", calc.calculateGrade(0));
    assertEquals("F", calc.calculateGrade(25));
    assertEquals("F", calc.calculateGrade(49));
}

@Test
public void testCalculateGrade_ClassD() {
    GradeCalculator calc = new GradeCalculator();
    assertEquals("D", calc.calculateGrade(50));
    assertEquals("D", calc.calculateGrade(55));
    assertEquals("D", calc.calculateGrade(59));
}

@Test
public void testCalculateGrade_ClassA() {
    GradeCalculator calc = new GradeCalculator();
    assertEquals("A", calc.calculateGrade(80));
    assertEquals("A", calc.calculateGrade(90));
    assertEquals("A", calc.calculateGrade(100));
}

@Test
public void testCalculateGrade_InvalidScore_ThrowsException() {
    GradeCalculator calc = new GradeCalculator();
    assertThrows(IllegalArgumentException.class,
        () -> calc.calculateGrade(-1));
    assertThrows(IllegalArgumentException.class,
        () -> calc.calculateGrade(101));
}
```

Test Exception

Kiểm tra xem method có ném đúng exception trong các tình huống lỗi hay không là **CỰC KỲ QUAN TRỌNG**. Đừng chỉ test "happy path"!

Kiểm tra exception bị ném

```
@Test  
public void testDivide_ByZero_ThrowsException() {  
    Calculator calc = new Calculator();  
  
    // Kiểm tra ném ArithmeticException  
    assertThrows(ArithmetiсException.class,  
        () -> {  
            calc.divide(10, 0);  
        }  
    );  
}
```

Kiểm tra exception message

```
@Test  
public void testWithdraw_InsufficientBalance() {  
    BankAccount account =  
        new BankAccount(100.0);  
  
    InsufficientBalanceException ex =  
        assertThrows(  
            InsufficientBalanceException.class,  
            () -> account.withdraw(200.0)  
        );  
  
    // Kiểm tra message  
    assertEquals("Insufficient balance",  
        ex.getMessage());  
  
    // Kiểm tra các thuộc tính khác  
    assertEquals(100.0, ex.getBalance());  
    assertEquals(200.0, ex.getAmount());  
}
```

- ☐ **Lưu ý:** Khi test exception, hãy kiểm tra cả **loại exception, message, và các thuộc tính** khác nếu có. Đừng chỉ kiểm tra có ném exception hay không!

Assert thay vì System.out.println()

COMMON MISTAKE

✗ Sai lầm phổ biến: Dùng System.out.println()

Nhiều bạn mới học thường viết test như thế này - đây là SAI HOÀN TOÀN:

✗ Code sai

```
public void testAdd() {  
    Calculator calc = new Calculator();  
    int result = calc.add(2, 3);  
  
    System.out.println("Result: " + result);  
    //
```

@BeforeEach và @AfterEach

TEST LIFECYCLE

Setup và Teardown

Khi nhiều test cần cùng một setup (khởi tạo objects, chuẩn bị dữ liệu), bạn có thể dùng **@BeforeEach** để tránh code trùng lặp:

```
public class BankAccountTest {  
    private BankAccount account;  
  
    @BeforeEach  
    public void setUp() {  
        // Chạy TRƯỚC MỖI test  
        account = new BankAccount(1000.0);  
        System.out.println("Setup completed");  
    }  
  
    @Test  
    public void testDeposit() {  
        account.deposit(500.0);  
        assertEquals(1500.0, account.getBalance());  
    }  
  
    @Test  
    public void testWithdraw() {  
        account.withdraw(300.0);  
        assertEquals(700.0, account.getBalance());  
    }  
  
    @AfterEach  
    public void tearDown() {  
        // Chạy SAU MỖI test  
        account = null;  
        System.out.println("Cleanup completed");  
    }  
}
```



Lưu ý: Mỗi test method sẽ có **instance mới** của test class. @BeforeEach đảm bảo mỗi test bắt đầu với state sạch sẽ, độc lập.

5.4. File I/O Cơ Bản

NIO.2 - MODERN API

Java NIO.2 (Java 7+)

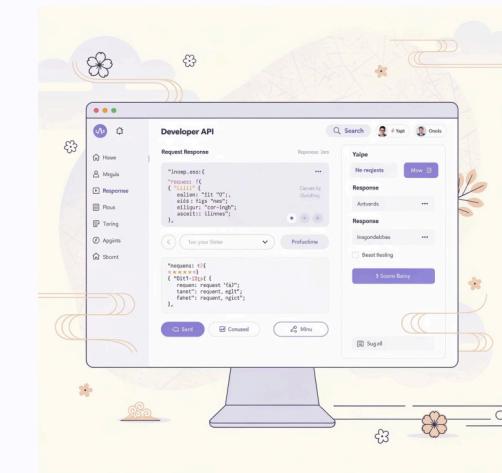
Java NIO.2 (New I/O version 2) là API hiện đại để làm việc với file, được giới thiệu từ Java 7. Nó đơn giản, mạnh mẽ và linh hoạt hơn nhiều so với API cũ (File class).

Import cần thiết

```
import java.nio.file.Path;  
import java.nio.file.Paths;  
import java.nio.file.Files;  
import java.nio.file.StandardOpenOption;  
import java.io.IOException;
```

3 class chính

- **Path:** Đại diện cho đường dẫn file/folder
- **Paths:** Factory class để tạo Path
- **Files:** Utility class với static methods để thao tác file



Path và Paths

Path là interface đại diện cho đường dẫn đến file hoặc directory trong file system:

```
// Cách 1: Dùng Paths.get() (Java 7+)
Path path1 = Paths.get("file.txt"); // Relative path
Path path2 = Paths.get("/home/user/file.txt"); // Absolute path (Linux/Mac)
Path path3 = Paths.get("C:", "Users", "user", "file.txt"); // Windows

// Cách 2: Dùng Path.of() (Java 11+) - Đơn giản hơn
Path path4 = Path.of("file.txt");
Path path5 = Path.of("/home/user/file.txt");

// Các method hữu ích của Path
System.out.println(path1.getFileName()); // file.txt
System.out.println(path1.getParent()); // null (vì là relative)
System.out.println(path2.getParent()); // /home/user
System.out.println(path1.toAbsolutePath()); // Full path
System.out.println(path1.isAbsolute()); // false
```

- ☐  **Lưu ý:** Path chỉ là đối tượng đại diện cho đường dẫn. Nó **KHÔNG** kiểm tra xem file có tồn tại hay không. Để thao tác với file thực sự, dùng class **Files**.

Files - Đọc File

Class Files cung cấp nhiều method tiện lợi để đọc file. Dưới đây là các cách phổ biến nhất:

```
Path path = Paths.get("file.txt");

try {
    // Cách 1: Đọc toàn bộ file thành String (Java 11+)
    String content = Files.readString(path);
    System.out.println(content);

    // Cách 2: Đọc thành List (mỗi dòng một phần tử)
    List lines = Files.readAllLines(path);
    for (String line : lines) {
        System.out.println(line);
    }

    // Cách 3: Đọc thành byte[] (cho binary files)
    byte[] bytes = Files.readAllBytes(path);

    // Cách 4: Đọc từng dòng với Stream (tiết kiệm bộ nhớ cho file lớn)
    try (Stream stream = Files.lines(path)) {
        stream.forEach(System.out::println);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

☞ **Best Practice:** Dùng `readString()` hoặc `readAllLines()` cho file text nhỏ. Dùng Stream (`Files.lines()`) cho file lớn để tránh hết bộ nhớ.

Files - Ghi File

Ghi file cũng rất đơn giản với class Files:

```
Path path = Paths.get("output.txt");

try {
    // Cách 1: Ghi String (Java 11+)
    String content = "Hello, World!\nThis is line 2.";
    Files.writeString(path, content);

    // Cách 2: Ghi List
    List lines = Arrays.asList("Line 1", "Line 2", "Line 3");
    Files.write(path, lines);

    // Cách 3: Ghi byte[]
    byte[] data = "Hello".getBytes();
    Files.write(path, data);

    // Cách 4: Append (thêm vào cuối file)
    Files.writeString(path, "\nNew line", StandardOpenOption.APPEND);

    // Cách 5: Ghi với options khác
    Files.writeString(path, content,
        StandardOpenOption.CREATE,      // Tạo mới nếu chưa tồn tại
        StandardOpenOption.TRUNCATE_EXISTING // Xóa nội dung cũ
    );

} catch (IOException e) {
    e.printStackTrace();
}
```

- StandardOpenOption.APPEND

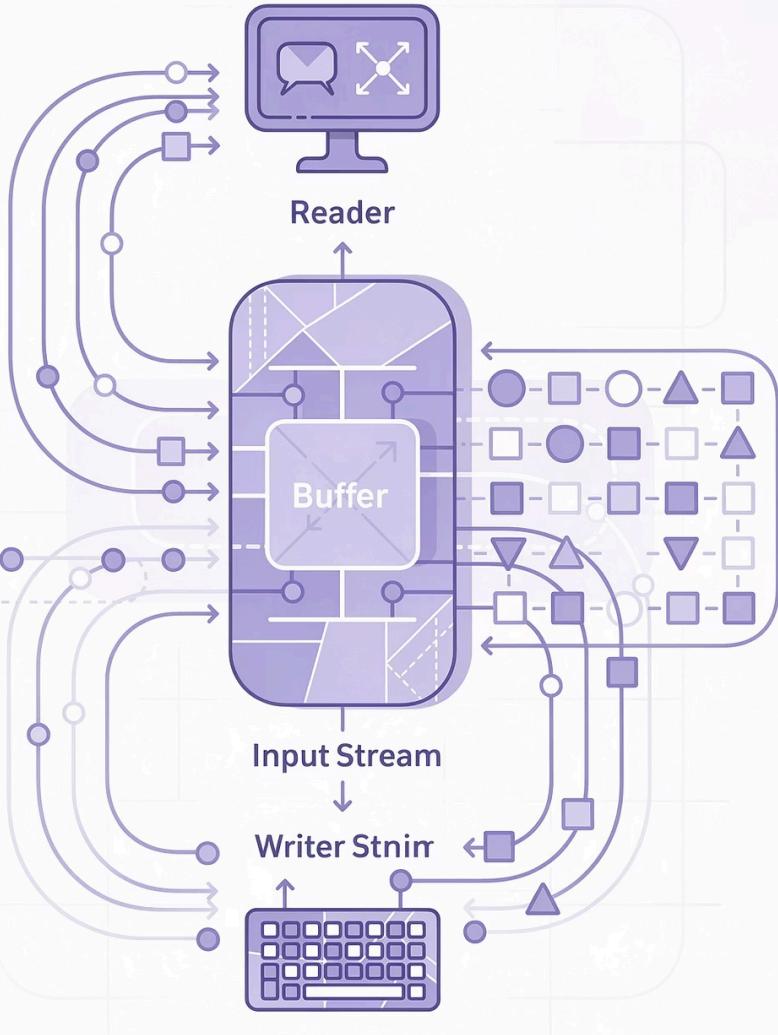
Thêm vào cuối file

- StandardOpenOption.CREATE

Tạo mới nếu chưa tồn tại

- StandardOpenOption.TRUNCATE_EXISTING

Xóa nội dung cũ



Try-with-Resources với I/O

Khi cần xử lý file lớn hoặc đọc/ghi từng dòng, dùng BufferedReader/BufferedWriter với try-with-resources:

Đọc file với BufferedReader

```
Path path = Paths.get("file.txt");

try (BufferedReader reader =
    Files.newBufferedReader(path)) {

    String line;
    while ((line = reader.readLine())
        != null) {
        System.out.println(line);
    }

} catch (IOException e) {
    e.printStackTrace();
}

// Reader tự động close
```

Ghi file với BufferedWriter

```
Path path = Paths.get("output.txt");

try (BufferedWriter writer =
    Files.newBufferedWriter(path)) {

    writer.write("Line 1");
    writer.newLine();
    writer.write("Line 2");
    writer.newLine();

} catch (IOException e) {
    e.printStackTrace();
}

// Writer tự động close và flush
```

- ❑ **Performance Tip:** BufferedReader/Writer sử dụng buffer nội bộ để giảm số lần truy cập disk, giúp cải thiện hiệu suất đáng kể khi xử lý file lớn.

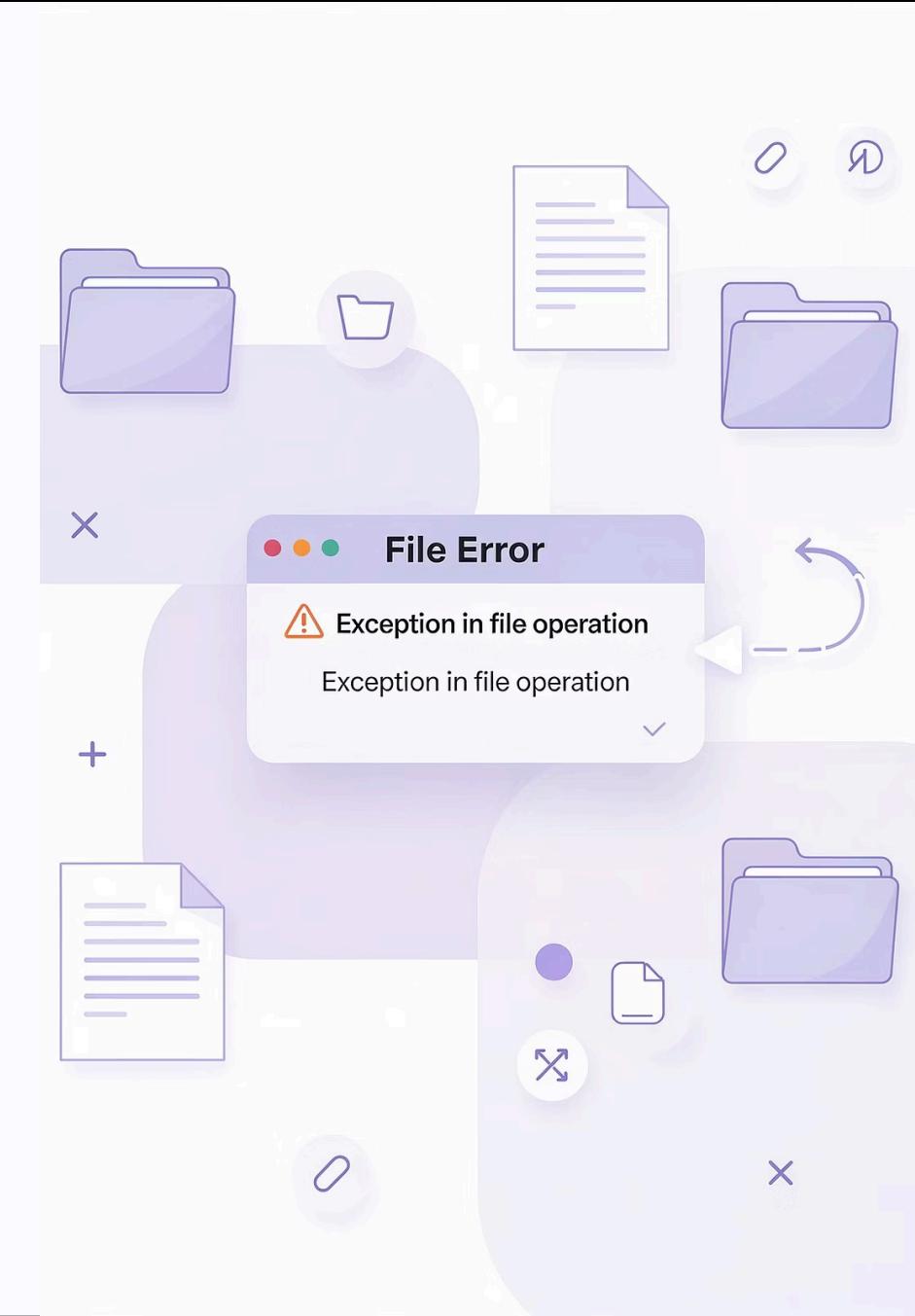
Xử lý Exception trong I/O

Best Practices khi làm việc với File I/O

Luôn dùng try-with-resources

1

```
//
```



5.5. Advanced I/O: Object Serialization

LƯU TRỮ OBJECTS

Serialization là gì?

Serialization (Tuần tự hóa) là quá trình chuyển đổi trạng thái của một Object thành byte stream để có thể:



Lưu vào file

Lưu trạng thái của object vào disk để dùng lại sau (Persistence)



Gửi qua mạng

Truyền object giữa các máy tính hoặc microservices (Network)



Lưu vào Database

Lưu object dạng binary vào cột BLOB trong database

Deserialization là quá trình ngược lại: chuyển byte stream trở lại thành Object.

Interface Serializable

Để một class có thể serialize, nó phải implements interface **java.io.Serializable**. Đây là **Marker Interface** (không có method nào):

```
import java.io.Serializable;

public class Student implements Serializable {
    private static final long serialVersionUID = 1L; // Version control

    private String name;
    private int age;
    private transient String password; // transient: KHÔNG serialize

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

    // Getters, setters...
}
```

serialVersionUID

ID để đảm bảo phiên bản class khi đọc/ghi là khớp nhau. Nếu không khớp → InvalidClassException.

transient

Từ khóa đánh dấu biến **KHÔNG** muốn lưu (ví dụ: mật khẩu, dữ liệu tạm). Khi deserialize, giá trị sẽ là default.

Ghi và Đọc Object

Sử dụng ObjectOutputStream để ghi (serialize) và ObjectInputStream để đọc (deserialize) object:

Ghi Object (Serialization)

```
Student s1 = new Student(  
    "Nam", 20, "secret123");  
  
try (ObjectOutputStream oos =  
        new ObjectOutputStream(  
            new FileOutputStream(  
                "student.dat"))) {  
  
    oos.writeObject(s1);  
    System.out.println(  
        "Đã lưu student!");  
  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Đọc Object (Deserialization)

```
try (ObjectInputStream ois =  
        new ObjectInputStream(  
            new FileInputStream(  
                "student.dat"))) {  
  
    Student s2 =  
        (Student) ois.readObject();  
  
    System.out.println(  
        "Đã đọc: " + s2.getName());  
    // password sẽ là null (transient)  
  
} catch (IOException  
        | ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

- ⚠ **Cảnh báo:** Serialization mặc định của Java có nhiều vấn đề về bảo mật và hiệu năng. Trong các dự án hiện đại (Web/API), người ta thường dùng **JSON** (Gson, Jackson) thay vì Serialization. Tuy nhiên, hiểu về nó vẫn cần thiết!

Tóm Tắt Chương 5

ON TẬP NHANH

Kiến thức đã học

Exception Handling

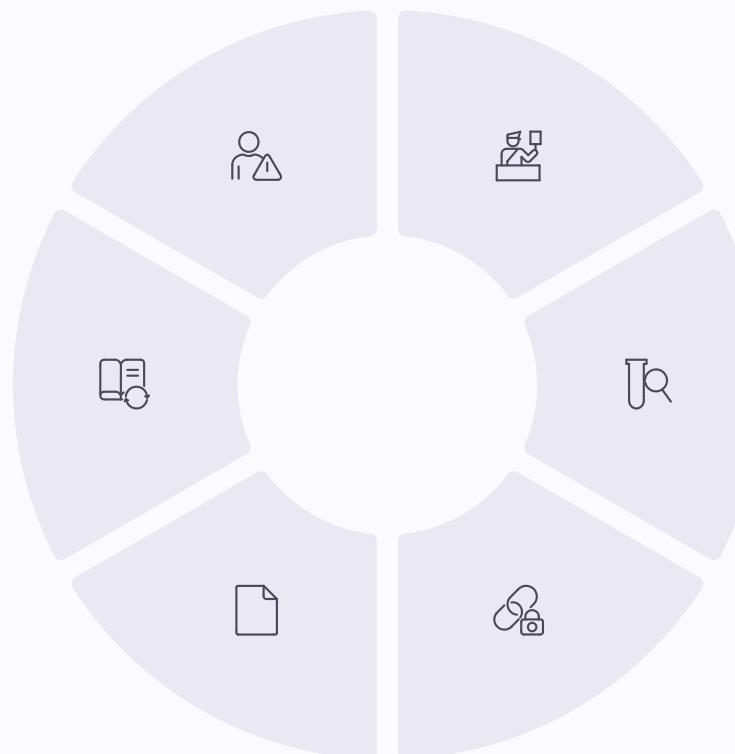
Phân biệt Checked vs Unchecked, sử dụng try-catch-finally và try-with-resources đúng cách

Serialization

Lưu trữ và khôi phục object với Serializable interface

File I/O

Đọc/ghi file với Java NIO.2, sử dụng Path, Files, BufferedReader/Writer



Custom Exception

Tạo exception riêng cho business logic với thông tin chi tiết

Unit Testing

Viết Unit Test chuyên nghiệp với JUnit 5, áp dụng AAA pattern

Testing Techniques

Boundary Testing và Equivalence Partitioning để tìm bugs hiệu quả



Kỹ năng đã có

100%

Exception Mastery

Xử lý lỗi đúng cách, không bao giờ "nuốt" exception

100%

Testing Pro

Viết Unit Test chuyên nghiệp, tự tin refactor code

100%

File I/O Expert

Thao tác file an toàn với NIO.2 và try-with-resources

100%

Best Practices

Áp dụng các kỹ thuật testing và error handling hiện đại

- ❑ **Chúc mừng!** Bạn đã hoàn thành chương quan trọng nhất trong hành trình trở thành Java Developer chuyên nghiệp. Exception Handling và Testing là **skills bắt buộc** mà mọi công ty đều yêu cầu. Hãy thực hành thật nhiều!



Bài Tập Chương 5

🎯 THỰC HÀNH

Bài 1: Exception Handling cơ bản

Yêu cầu

Tạo class Calculator với method divide(int a, int b):

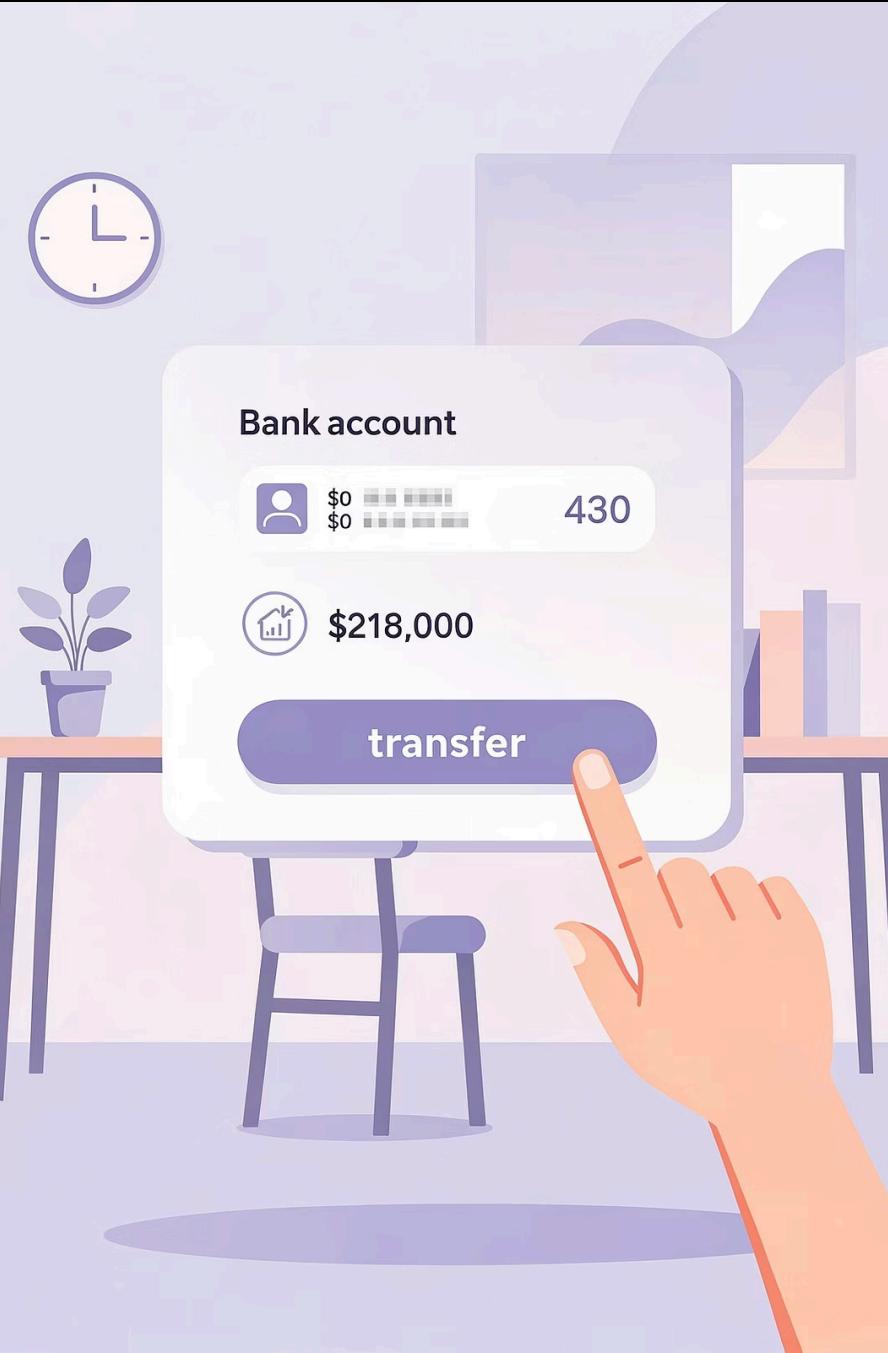
1. Ném ArithmeticException nếu $b = 0$
2. Viết method test với try-catch
3. In thông báo lỗi thân thiện cho user
4. Viết Unit Test để kiểm tra exception

Test Cases

```
Calculator calc = new Calculator();
```

```
// Test case 1: Normal  
calc.divide(10, 2);  
// Kết quả: 5
```

```
// Test case 2: Exception  
calc.divide(10, 0);  
// Exception với message  
// "Cannot divide by zero!"
```



Bài 2: Custom Exception

Yêu cầu

1. Tạo custom exception **InsufficientBalanceException** với thuộc tính balance và amount
2. Tạo class **BankAccount** với method withdraw(double amount)
3. Ném exception nếu số dư không đủ, kèm theo thông tin chi tiết
4. Viết Unit Test đầy đủ cho exception case

Test Cases

```
BankAccount account = new BankAccount(100.0);
account.withdraw(50.0); //
```

Bài 3 & 4: File I/O và Unit Test



Bài 3: Try-with-Resources

Yêu cầu:

1. Đọc file "input.txt"
2. Xử lý dữ liệu (ví dụ: uppercase mỗi dòng)
3. Ghi kết quả vào "output.txt"
4. Sử dụng try-with-resources
5. Xử lý tất cả exceptions

Bài 4: Unit Test với JUnit 5

Yêu cầu:

Tạo class StringUtils với methods:

1. reverse(String str): Đảo ngược chuỗi
2. isPalindrome(String str): Kiểm tra đối xứng
3. countWords(String str): Đếm số từ

Viết Unit Test cho:

- Các trường hợp bình thường
- Boundary cases (null, empty)
- Exception cases

Bài 5, 6, 7: Tổng hợp

Bài 5: Boundary Testing

Tạo class GradeCalculator với method calculateGrade(double score) theo quy tắc: 0-49 = "F", 50-59 = "D", 60-69 = "C", 70-79 = "B", 80-100 = "A".

Viết test cho:

- Các giá trị biên (0, 49, 50, 59, ...)
- Giá trị ngoài phạm vi (< 0, > 100)

Bài 6: File I/O

Tạo class FileManager với methods:
readFile(String filename), writeFile(String filename, String content),
appendToFile(String filename, String content).

Yêu cầu:

- Xử lý exceptions đầy đủ
- Viết Unit Test
- Dùng try-with-resources

Bài 7: Hệ thống File Service

Tạo hệ thống quản lý file hoàn chỉnh với class FileService (readFile, writeFile, copyFile), custom exception FileServiceException, và Unit Tests đầy đủ.

Mục tiêu: Test coverage cao, xử lý tất cả edge cases, code chuyên nghiệp.

Chúc Bạn Học Tốt! 🚀

📚 Tài Liệu Tham Khảo

JUnit 5 User Guide

Tài liệu chính thức đầy đủ nhất về JUnit 5

<https://junit.org/junit5/docs/current/user-guide/>

Java NIO.2 Tutorial

Hướng dẫn chi tiết về File I/O với NIO.2 từ Oracle

<https://docs.oracle.com/javase/tutorial/essential/io/fileio.html>

Effective Java

Cuốn sách kinh điển của Joshua Bloch - Item 69-77: Exceptions

Must-read cho mọi Java developer!

- ⚠️  **Lời khuyên cuối cùng:** Exception Handling và Testing là những kỹ năng **PHÂN BIỆT** giữa developer nghiệp dư và chuyên nghiệp. Hãy thực hành đều đặn mỗi ngày. Viết test cho mọi method bạn tạo ra. Xử lý exception một cách có trách nhiệm. Đây chính là con đường dẫn đến thành công!

Chúc bạn thành công trên hành trình trở thành Java Developer chuyên nghiệp! 🎓✨