Exceptions and File-based Input/Output
kapitel 14

Joakim Iversen

January 15, 2025

Disposition:

1. Motivation
2. Exceptions
3. Defensiv Programmering
4. File-Based Input/Output

Noter

Motivation

Det er umuligt at programmere et program, uden på et eller andet tildspunkt undervejs at lave en fejl. Ugyldige parametre, en manglende fil eller brugergenerede problemer. Det er derfor afgørende, at vi som programmører ikke bare håndtere de fejl der må opstå under programmeringen, men også forebygger eventuelle fejl efter produktionen.

Og det er her Exceptions, defensiv programmering kommer ind i billedet. Samt at jeg også vil gennemgå Fil basseret Input og Output.

Exceptions

Exceptions hjælper os med at håndtere en givet fejl i et program. Hvis vi tager brugerinput er en meget almindelig fejl, at brugeren giver os et input som er **null** eller tom **("")**. Her er det muligt for os at bruge exceptions til at sikre os at programmet stadig fungere, trods det ugyldige input, og ikke crasher.

Når vi arbejder med exceptions kører de på en **try** og **catch** blok. Disse fungere ved at den først kører koden i **try** blokken, hvis der så opstår en fejl her tjekker den om den givet fejl er blevet "grebet" i vores **catch** blok.

```
filename = ... // Et eller andet brugerinput
boolean sucfcesful = false
do {
    try {
        contacts.saevToFile(filename);
        succesful = true;
    }
    catch (IOException e) {
        System.out.println(...);
        filename = ... // Ny forespørgelse
    }
} while (!succesful);
```

Læg mærke til vores try-catch blok er inde i en do-while løkke der bliver ved med at forespørge et filnavn indtil det bliver givet et gyldigt resultat.

Defensiv Programmering

Når vi snakker om defensiv programmering snakker vi om, at sikre at programmet selv tjekker for gyldigheden af inputtet. Vi kan her kigge på metoden removeDetails fra AddressBook:

```
public boolean removeDetails(String key) {
    if (key == null) {
        throw new IllegalArgumentException("...");
    }
    if (keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.reamove(details.getName());
        book.remove(details.getPhone());
        return true;
    } else {
        return false;
    }
}
```

Her kan vi se at metoden ikke vil prøve og behandle en ugyldig nøgle, da det vil falde i det første if-statement eller i else-blokken i det andet. Vi bruger IllegalArgumentException til at kunne tydliggører hvad fejlen er, og hvor det er den opstår.

Filbaseret input/output

Når vi begynder at arbejde med filer træder vi ind på et emne, hvor der hurtigt kan opstå diverse fejl. En fil kan mangle, man har utilstrækkelige tilladelser eller noget helt tredje. I Java er der effektive metoder til at håndtere sådanne fejl. Vi vil kigge på et eksempel til at læse fra en fil:

```
Charset charset = Charset.forName("US-ASCII");
ath path = Paths.get("...");

try (BufferedReader reader = Files.newBufferedReader(path, charset)) {
    String line = reader.readLine();
    while (line != null) {
        System.out.println(line);
    }
}
catch (FileNotFoundException e) {
    System.out.println("...");
}
catch (IOException e) {
    System.out.println("...");
}
```

Læg mærke til at vi kan gribe flere forskellige exceptions, ved bare at "stacke" vores catch-blokke.

Vi åbner filen ved hjælp af en klassemetode i Files klassen. Vi placere åbningen som paramtre i try parantessen, da man så sikre sig at filen automatisk lukkes efter at den er blevet læst.

Vi kan også se at vi burger to forskellige exceptions. Dette er da vores IOException (Input/output Exception) er den generelle klasse for fejl der sker ved håndtering af input/output. Vi vil derfor godt kunne catche denne fejl alene, men ved at bruge FileNotFound bliver vi mere præcise. Det er af samme grund vigtigt at tjekke for FileNotFoundException først, da denne exception også vil blive grebet af IOException.