Kivételkezelés és naplózás

Simon Károly
simon.karoly@codespring.ro

Try-catch

```
try {
      // Kódrészlet, amely kivételt eredményezhet
} catch (Exception1 object1 ) {
      // Az Exception1 kivétel kezelésének megfelelő kód
} catch (Exception2 object2 ) {
      // Az Exception2 kivétel kezelésének megfelelő kód
} finally {
      // Ez a kódrészlet minden esetben végre lesz hajtva
}
```

- Egy kivétel felléptekor az adott kivétel típusának megfelelő első catch ágon belüli kód kerül lefuttatásra.
- Ha a figyelt kivételtípusok egymás leszármazottjai fontos a catch ágak sorrendje!
- A finally ág utasításai minden esetben végrehajtásra kerülnek (pl. nem kezelt kivétel előfordulása, vagy kilépés (pl. return) esetén is) → erőforrások felszabadítása, kapcsolatok zárása

Kivételkezelés példa

```
public class ExceptionExample {

        public static void main (String[] args) {
                int i;
                try {
                        i = Integer.parseInt (args[0]);
                } catch (ArrayIndexOutOfBoundsException e1) {
                        i = 10;
                } catch (NumberFormatException e2) {
                        i = 20;
                } finally {
                        i++;
                }
                System.out.println(i);
        }

}
```

Throwable

- Throwable osztály: a Java kivételosztályok hierarchiájának csúcsa.
- Throwable objektumok: a kivételek közös tulajdonságainak rögzítésére és lekérdezésére → "pillanatkép" az objektumhoz tartozó végrehajtási szálhoz tartozó hívási veremről, a kivétel fellépésének pillanatában.
- Információk lekérdezése: getStackTrace metódus (kiírás printStackTrace).
- Lekérdezés eredménye: StackTraceElement típusú elemeket tartalmazó tömb.
- Lekérdezhető információk: metódus és osztály neve, forráskód állomány neve, a megfelelő sor száma stb. + szöveges információ a kivételről.
- Láncolt kivételek (chained exceptions): a kivétel valamilyen másik kivétel következménye, az "ok" (a másik kivételre mutató referencia) lekérdezhető a getCause metódussal.
- Három kivételtípus: ellenőrzött (checked), futási idejű (runtime), hiba (error). Az utóbbi két kategóriát együttesen nem ellenőrzött (unchecked) kivételeknek is nevezik.

Ellenőrzött kivételek

- Előfordulásuk előrelátható, az alkalmazástól elvárhatjuk, hogy megfelelően kezelje ezeket.
- Példa: állománykezeléssel kapcsolatos műveletnél a felhasználótól egy állomány nevét kérjük. Általában létező állomány nevét adja meg, de előfordulhat, hogy ez nem így történik. A hibalehetőségre számíthatunk, elvárható, hogy megfelelően kezeljük, és a rendszer is ezt várja el tőlünk → az ilyen típusú kivételeket kötelező módon kezelnünk kell.
- Kivételek kezelése: try-catch alkalmazása (a kivételt "helyben" kezeljük), vagy a kivétel "továbbdobása", a metódus fejlécében jelezve, hogy meghívása kivételt eredményezhet (a kivételt az alkalmazás más, felsőbb rétegei felé küldjük). A második esetben egy felsőbb rétegben mindenképpen kötelező lesz a try-catch alkalmazása, ellenkező esetben fordítási hibát kapunk.
- Az ellenőrzött kivételeknek megfelelő osztályok közös őse az **Exception** osztály (a Throwable leszármazottja).
- Példák: FileNotFoundException, IOException, stb.

Futási idejű kivételek

- Szintén az alkalmazással kapcsolatos belső körülmények következtében lépnek fel, de előfordulásuk nem, vagy nehezebben előrelátható.
- Általában programozási hibák, bug-ok eredményei → a helyes megoldás ezeknek a hibáknak a kiszűrése.
  - Példa: állománykezeléssel kapcsolatos műveletet végző metódusnak null értékű állománynevet adunk át paraméterként, és NullPointerException kivételt kapunk.
  - Lehetőségünk van a kivétel kezelésére, de jobb/biztosabb megoldás lehet a hiba forrásának kiküszöbölése.
- Példák: szöveges adatok numerikusba alakítása (NumberFormatException), közvetlen hivatkozás tömbök elemeire (ArrayIndexOutOfBoundsException).
  - A rendszer nem várhatja el minden ilyen kivétel kezelését (pl. mindig try-catch szerkezetet kellene alkalmazni, amikor tömbök elemeire hivatkozunk) → nem kötelező a try-catch alkalmazása (a metódusok meghívhatóak, nem kapunk fordítási hibát).
- Az ilyen kivételek nehezebben azonosíthatóak, és potenciálisan futási idejű hibák forrásai lehetnek (figyeljünk!).
- A futási idejű kivételeknek megfelelő osztályok közös őse a **RuntimeException** osztály (az Exception leszármazottja).

Hibák

- Az alkalmazástól független külső körülmények hatására lépnek fel, általában "komoly" következményekkel járnak, és előfordulásuk nem előrelátható.
- Kezelésük értelmetlen, mert az alkalmazás általában olyan állapotba kerül, ahonnan már nem térhet vissza a normál működéshez.
- Általában a helyes megoldás a program lezárása.
- Példák: virtuális gép meghibásodása, vagy erőforráskeret túllépése (VirtualMachineError), szükséges osztály definíciója nem betölthető (NoClassDefFoundError) stb.
- Az ilyen problémákat okozó hibatípusok száma a másik két kategóriához viszonyítva kicsi (szerencsére).
- Az ilyen típusú kivételeknek megfelelő osztályok közös őse az **Error** osztály (a Throwable osztály leszármazottja)

Kivételek továbbítása

- Példa:

```
public void readFile (String filename)
                        throws FileNotFoundException, IOException {
   FileReader fr = new FileReader (filename);
   BufferedReader br = new BufferedReader (fr);
   String line;
   while ((line = br.readLine ()) != null) {
                System.out.println (line);
      }
}
```

- **Recept:** az alsó rétegeken belül naplózzuk a kivételeket, majd a felsőbb rétegek fele küldjük tovább azokat, és a legfelső szinten (prezentációs réteg, grafikus felület) kezeljük (megfelelő üzenetet jelenítve meg a felhasználónak).
- Példa: ha csak szerver oldalon kezelünk egy adatbázis hozzáféréssel kapcsolatos kivételt, nem jut el az információ a felhasználóhoz. Szerver oldalon viszont fontos lehet a naplózás (logging) a hibaelhárítási, javítási műveletek támogatásához.

Saját kivételek

- Szükségünk lehet saját, pl. speciális eseteknek megfelelő, speciális információkat rögzítő kivételtípusokra.
- Kivételosztály létrehozása: származtatás valamelyik alaposztályból (pl. Exception):

```
public class MyException extends Exception {
     private String info;
     public MyException (final String info) {
          super (info);
          this.info = info;
     }
     public String getInfo () {
          return info;
     }
}
```

Saját kivételek - recept

- Az egyszerű felhasználók sok esetben nem érdekeltek a hiba pontos típusában (a részletkérdések számukra nem feltétlenül fontosak).
- **Recept:** szerver oldalon kezeljük és naplózzuk a kivételeket, pontos típusuknak megfelelően, de a felsőbb rétegekhez saját kivételobjektumokat továbbítunk.
- Példa: keretrendszeren belüli különböző kivételtípusok "egybeolvasztása" (pl. SQLException) → egységes kezelési lehetőség a felsőbb rétegeken belül.
- Megjegyzés: az ilyen esetekben fontos lehet a kivételek "láncolása".
- Láncolt kivételeket alkalmazhatunk a Throwable(String message, Throwable cause), vagy Throwable(Throwable cause) konstruktorok segítségével (pl. a MyException példánk konstruktorát módosíthatjuk a megfelelő módon).
- "Dilemma": a saját kivételek az Exception vagy a RuntimeException osztályból származzanak. A döntés helyzet-specifikus, de több esetben jobb megoldás lehet nem "kötelezni" a felsőbb rétegeket megvalósító fejlesztőket a kezelésre (csak indokolt esetben).

Saját kivételek

```
public void readFile (String filename) throws MyException {
     try {
          FileReader fr = new FileReader (filename);
          BufferedReader br = new BufferedReader (fr);
          String line;
          while ((line = br.readLine()) != null) {
               System.out.println (line);
          }
        } catch (FileNotFoundException ex1) {
          //... - log this error
          throw new MyException ("data access error");
        } catch(IOException ex2) {
          //... - log this error
          throw new MyException ("data access error");
        }
    }
```

Java SE 7 újdonságok

- Három fontos újítás/módosítás a kivételkezeléssel kapcsolatban:
  - Lehetőség több kivételtípus kezelésére egy catch ágon belül.
  - Hatékonyabb típusellenőrzés a kivételek továbbdobásánál.
  - Egyszerűsített erőforrás felszabadítás.
- Több kivételtípus egy catch ágon belül:

```
try {
  …          //call some methods with db and file I/O operations
} catch (IOException | SQLException ex) {
  …          //exception handling
}
```

- Megjegyzés: a hasonló esetek közös ősosztály segítségével történő kezelése (pl. Exception típus) nem szerencsés megoldás. Pl. megjelenhet egy más típusú kivétel a try blokkon belül, amelyet nem lenne jó ötlet ugyanúgy kezelni, de az Exception alkalmazása "elrejti" előlünk a problémát → hibák/bajok forrása lehet (pl. a debugging folyamatot megnehezítve).

Java SE 7 újdonságok

- Hatékonyabb típusellenőrzés a továbbdobás esetében:
  - Akkor működik, ha nem történik megfeleltetés művelet a catch blokk paraméterével kapcsolatban (a paraméter final) (final rethrow feature).
  - Lényege: ha a kapott kivétel a catch paraméter típusának ősosztálya vagy leszármazottja, akkor a konkrét típusnak megfelelően lesz továbbdobva (nem a catch paraméter típusa alapján).
  - Alkalmazás példa: több kivételtípus általános/összevont kezelése helyben és lehetőség egy "precízebb" (konkrét típusnak megfelelő) kezelésre máshol (felsőbb rétegekben).
  - Pl. az alábbi kódrészlet nem fordítható le a 7-es előtti verziókkal:

```
private void exceptionExample () throws IOException {
        try {
            throw new IOException ("Error");
        } catch (Exception exception) {
            throw exception;
        }
    }
```

Java SE 7 újdonságok

- Egyszerűsített erőforrás-felszabadítás:

```
try (BufferedReader br = new BufferedReader (new FileReader("somefile.txt"))) {
      …
} catch (IOException ioe) {
       System.err.println (ioe.getMessage());
}
```

- A Closeable vagy AutoCloseable interfészt megvalósító erőforrások automatikus lezárása → nincs szükség a finally ágon belüli lezárásra (ettől még lehet finally ág is).
- A catch és finally ágakon belüli utasítások a lezárások után fognak lefutni.
-
```
static void copy(String srcFile, String dstFile) throws IOException {
        try (FileInputStream fins = new FileInputStream(srcFile);
             FileOutputStream fouts = new FileOutputStream(dstFile)) {
           …    //I/O operations
        }
 }
```

- A lezárás sorrendje: `fouts.close (), fins.close ()`

Java SE 7 újdonságok

- Egyszerűsített erőforrás-felszabadítás – megjegyzések:
  - Closeable vs. AutoCloseable:
    - A Cloaseable az AutoCloseable leszármazottja.
    - A close metódus IOException-t dobhat az első esetében, Exception-t a másodikában (a konkrét implementációk általában specifikusabb kivételt dobnak).
    - A close többszöri meghívásának nincs hatása az első esetben, lehet hatása a másodikában (bár ajánlott olyan módon megvalósítani, hogy ne legyen).
  - A háttérben meghívott close metódusok által esetlegesen dobott kivételek "elfojtódnak" (suppress). Lekérdezésük a Throwable osztály getSuppressed metódusával lehetséges (Throwable tömböt térít vissza), az "elfojtás" implicit módon történik, az addSuppressed metódus segítségével.
  - Példa AutoCloseable/Closeable implementációkra: ObjectInput, ObjectOutput, Connection, Statement, ResultSet stb.

Naplózás

- Naplózási (logging) műveletek: Logger objektumok metódusainak meghívása.
- A Logger objektumok LogRecord objektumokat tartanak nyilván, amelyeket Handler objektumok segítségével "publikálnak".
- Fontossági szint (Level): az üzenet fontossága, prioritása (egy egész érték).
- A Level osztály (Java logging API) standard értékei: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST.
- Szürők (Filter) alkalmazásának lehetősége.
- Formázási lehetőségek (Formatter alkalmazása).
- A Logger-ek (általában) névvel (pl. "java.awt") rendelkeznek, a hierarchikus névteret a LogManager kezeli, minden Logger nyilvántartja az "ősét" (és örökölhet tőle bizonyos tulajdonságokat).
- Standard Handler-ek (Java logging API): StreamHandler, ConsoleHandler, FileHandler, SocketHandler, Memory Handler.
- Standard Formatter-ek: SimpleFormatter, XMLFormatter.
- Konfigurációs állomány alkalmazható.

Naplózás példa

```
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.LogManager;
import java.util.logging.Logger;
import java.util.logging.XMLFormatter;

public class LoggerExample {

    public static void main (String[] args) {
        FileHandler fh = null;
        try {
            LogManager lm = LogManager.getLogManager ();
            Logger logger = Logger.getLogger ("LoggerExample");
            lm.addLogger (logger);
            fh = new FileHandler ("log_test.txt");
            fh.setFormatter (new XMLFormatter ());
            logger.addHandler (fh);
            logger.setLevel (Level.INFO);
            logger.log (Level.INFO, "test 1");
            logger.log (Level.INFO, "test 2");
        } catch (Exception e) {
            e.printStackTrace ();
        } finally {
            if (fh != null) {
                fh.close ();
            }
        }
    }
}
```

Log4j

- Java naplózási keretrendszer, Apache Software Foundation.
- Szintek: FATAL, ERROR, WARN, INFO, DEBUG, TRACE.
- Több Handler: JDBCAppender, JMSAppender stb., stb.
- Konfiguráció: XML, vagy .properties
- Konfiguráció példa: log4j.properties (a projekt gyökérkönyvtárában)

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
log4j.rootLogger=debug, stdout
```

- Példa:

```
import org.apache.log4j.Logger;
public class LoggingExample2 {
    private static Logger log = Logger.getLogger(LoggingExample2.class);
    public static void main(String[] args) {
        log.trace("Trace");
        log.debug("Debug");
        log.info("Info");
        log.warn("Warn");
        log.error("Error");
        log.fatal("Fatal");
    }
}
```

slf4j

- Különböző naplózási keretrendszerek fölötti absztrakciós szintet képez.
- A fejlesztőknek lehetősége van a telepítéskor "bekötni" a kívánt naplózási rendszert.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
  public static void main(String[] args) {
    Logger logger = LoggerFactory.getLogger(HelloWorld.class);
    logger.info("Hello World");
  }
}
```

- slf4j-api jar állomány szükséges, ezen kívül valamilyen konkrét naplózási keretrendszernek megfelelő csomag (binding). Példa: slf4j-simple jar.
- log4j használata esetén szükséges az slf4j-log4j binding-nak megfelelő jar, valamint a log4j csomag, beállítások (properties)
- További támogatott keretrendszerek: java.util.logging, jcl stb.