第6章. 適切な例外処理を書こう

読みやすいプログラムを書くには、適切な例外処理を書くことが大切です。
適切な例外処理が記述されていれば、例外が発生した時に、その例外からエラーの原因や解析が可能になるため、問題が起きた時の対処がしやすくなります。
この章では、例外クラスを使う上での気をつけるポイントについて説明します。

6－1. ストリームの close 処理は finally 句で行おう

データベースを「open」した場合は、必ず finally 句で close メソッドを呼び出して close 処理を行なうようにしましょう。

close 処理をしなくても、最終的にはデータベースの接続はガーベージコレクタによって切断されますが、close 漏れによりメモリ不足を起こします。
close 処理は何か問題が発生した時にも忘れずに行なうために、次のように finally 句を使って記述します。

finally句を使用した例

```
/**
 * 顧客情報削除.
 *<br>
 *
 * @param id 顧客マスタID.
 */
public void remove(String id) {

    DataBaseAccess dataBaseAccess = new DataBaseAccess();

    try {

        // DB接続
        dataBaseAccess.open();

        // TB_CUSTOMERテーブルから削除
        String sql = "DELETE FROM TB_CUSTOMER WHERE id = ?";

        :
        :
    } catch (SQLException e) {
        e.printStackTrace();

    } finally {

        // DB切断
        dataBaseAccess.close();
    }
}
```

finally句でcloseメソッドを呼び出して確実にclose処理を行なっている。

６－２. 使用用途に合わせて、例外クラスを適切に使い分けよう

Exception クラスには、たくさんのサブクラスがあります。

catch 句に catch(Exception e) と書いてしまうと、Exception クラスを祖先にもつ全ての例外を捕捉することになります （Exception クラスを含めの全サブ（サブサブ・・・）クラス）。 必要以上に大きな範囲で捕捉するのは、余計な例外を拾って不正な対処を行なったり、例外の破棄につながります。 また、例外が発生したことは認識できますが、どのような例外が発生したのかが不明確です。

Exception の継承関係

```
Object
    └Throwable
        └Exception
                ├.....
                ├RuntimeException
                |       ├.....
                |       ├IllegalArgumentException
                |       |       ├.....
                |       |       ├NumberFormatException
                |       |       └.....
                |       └.....
                └.....
```

それぞれ例外として発生する理由が異なるので、発生しうる例外ごとにの catch 句を追加するべきです。

例えば、データベース関係の例外では SQLException を使用したり、 文字列を数値型に変換する時の例外では NumberFormatException 等を使用することがあります。

また、catch 句を複数記述する場合、投げられる例外オブジェクトに合わせて、プログラムは、ソースコードの上から下へ順序よく実行されます。 そのため、一つの catch 句が実行されるとそれ以降の catch 句は実行されないので、範囲が狭いものから先に記述する必要があります。

×：catch句の複数記述が適切ではないソースコード

```
public void changeError() {

    try {

        // 文字列を数値に変換
        Integer.parseInt("abc");

    } catch (Exception ex) {

        System.out.println("Exceptionが発生しました");

    } catch (NumberFormatException en) {

        System.out.println("NumberFormatExceptionが発生しました");
    }
}
```

範囲の広いExceptionが先に書かれているため、例外が発生した時に、NumberFormatExceptionの処理は実行されない。

○：catch句の複数記述が適切であるソースコード

```
public void changeError() {

    try {

        // 文字列を数値に変換
        Integer.parseInt("abc");

    } catch (NumberFormatException en) {

        System.out.println("NumberFormatExceptionが発生しました");

    } catch (Exception ex) {

        System.out.println("Exceptionが発生しました");
    }
}
```

範囲の狭い順に例外クラスが書かれているので、例外が発生した時に、2つ目のcatch句も実行される。

6－3. catch ブロックの処理で例外情報を出力・通知を記述しよう

catch ブロックの処理でシステム（開発者）に対して例外情報を出力する記述や、ユーザ（システムの利用者）に対してエラーが発生したことを通知する記述を書きましょう。

catch ブロックの処理が何も書かれていないと、何かしらの例外を catch した時、何も処理が行なわれないまま正常終了として呼び出し元に制御が戻ってしまいます。例えば、Web 上でユーザ ID やパスワードを入力して、ログインする時等、エラーが発生した場合、ログイン処理が失敗したということがユーザやシステムに適切に通知されません。
結果的に、何らかの障害が発生しているという事実が発覚するのが遅くなり、更に障害発生時の例外の情報が一切得られないため、その原因の解析が困難になってしまいます。

このような事態を避けるため、例外処理を正しく行なう必要があります。 例外を catch した時、以下のような処理を行ないます。

１．ログ出力（原因解析のための情報の出力）

エラーの発生箇所（メソッド）を特定できるようスタックトレース（e.printStackTrace();）を出力しよう。

通常の開発では、ログ出力を行うことになると思います。 その際のログ出力メソッドの引数に例外オブジェクト e を渡して、エラーメッセージと一緒にスタックトレース情報もログファイルに書き出されるようにするのが一般的です。そのあたりは各プロジェクトのログ出力方針に従って実装します。

２．呼び出し元へ例外通知（障害発生の通知）

「処理が失敗した」ということが最終的にユーザに伝わるよう、ログイン処理や顧客情報の操作を行なう等のサービスクラスで例外を catch したときは、そこで例外処理を完結させず、呼び出し元に例外が発生したことを通知する記述を書きましょう。

具体的には、拾った例外を throw し直します。

×：例外処理が適切ではないソースコード

```
/**
 * 顧客情報削除.
 *<br>
 *
 * @param id 顧客マスタID.
 */
public void remove(String id) {

    DataBaseAccess dataBaseAccess = new DataBaseAccess();

    try {

        // DB接続
        dataBaseAccess.open();

        // TB_CUSTOMERテーブルから削除
        String sql = "DELETE FROM TB_CUSTOMER WHERE id = ?";

        :
        :
    } catch (Exception e) {
    } finally {

        // DB切断
        dataBaseAccess.close();
    }
}
```

catchブロックで例外情報を出力する記述が書かれていないため、エラーが発生したことが、システムに適切に通知されない。
また、Exceptionなので、どのような例外が発生したかが分からない。

○：例外処理が適切であるソースコード

```
    /**
     * 顧客情報削除.
     *<br>
     *
     * @param id 顧客マスタID.
     * @throws Exception
     * @throw  ClassNotFoundException JDBCドライバ接続時の例外.
     * @throw  SQLException           SQL例外.
     */
    public void remove(String id) throws Exception {

        DataBaseAccess dataBaseAccess = new DataBaseAccess();

        try {

            // DB接続
            dataBaseAccess.open();

            // TB_CUSTOMERテーブルから削除
            String sql = "DELETE FROM TB_CUSTOMER WHERE id = ?";

            :
            :
        } catch (ClassNotFoundException | SQLException e) {

            // システム（開発者）に対して例外情報を出力
            e.printStackTrace();

            // 呼び出し元へエラー発生の通知
            throw e;

        } finally {

            // DB切断
            dataBaseAccess.close();
        }
    }
```

呼び出し元に発生した例外が投げられるようにするため、メソッドの宣言にはthrowsを記述する。

論理演算子|を使用すると、catch句を複数記述する処理と同じ意味になる。発生しうる例外ごとに書かれているため、どのよう例外が発生したかが明確。

e.printStackTrace();の記述により、エラー発生箇所を特定できる。

拾った例外をthrowし直して、呼び出し元にエラーが発生したことを適切に通知。呼び出し元では、最終的にはユーザに「処理が失敗したということが通知されるように、例外処理を行なう。