Object Oriented Programming 2

Rolf Haenni & Andres Scheidegger

Exercises 3

Introduction

In exercises 2 you have programmed an application which stores and loads person data to / from text files. Dealing with files (open, read, write) and parsing data from a text file may result in different exceptions and runtime exceptions. Our goal is to abstract these low-level exceptions by a few self-made high-level exceptions thrown by the persistence layer and to catch them in the application. The user shall then be informed through an Alert dialog. The application shall recover from the exceptional state and allow the user to continue his / her work.

1. Self-made Exception for Persistence Layer

Create a first self-made exception in your *persistence* package. Give it an appropriate name, e.g. *PersistenceException*[1]. Decide whether this exception should be checked or unchecked. Provide the necessary constructors to allow exception chaining. Add *throws* clauses to the *load()* and *save()* methods in the DAO interface.

2. Try-with-Resources and Re-throwing Exception

Refactor your DAO implementation for text files to use try-with-resource clauses in the load() and save() methods. Use exception chaining in the exception handlers for the *FileNotFoundException* and the *IOException*. Re-throw these exceptions wrapped into a *PersistenceException*.

3. Catch Exceptions in GUI (and in JUnit-Test)

In your GUI you must now deal with *PersistenceException* objects thrown by your persistence layer. Add the necessary *try-catch* statements in your action handlers for the *Load* and the *Store* button. When a *PersistenceException* occurs an Alert dialog shall be opened, which reports the problem to the user.

[1] Do not confuse our self-made PersistenceException with the one declared in the package javax.persistence (Java Enterprise Edition)

When trying to test your application, you will find it nearly impossible to provoke a file related exception with the *FileChooser*. This dialog already handles such exceptions internally and will provide your application with a valid file to read from / write to.

Hence, we have to use JUnit to test whether inappropriate files passed to the text file DAO result in a *PersistenceException* when calling *load()* or *store()*. E.g. you may use an empty file path (`new File("")`) to provoke this.

To check for an expected exception in a JUnit test you need to write the following annotation before the test method:

```
@Test(expected = PersistenceException.class)
public void testSaveWithIllegalFileName() throws PersistenceException {…}
```

4. Runtime Exceptions during Parsing of Person Data

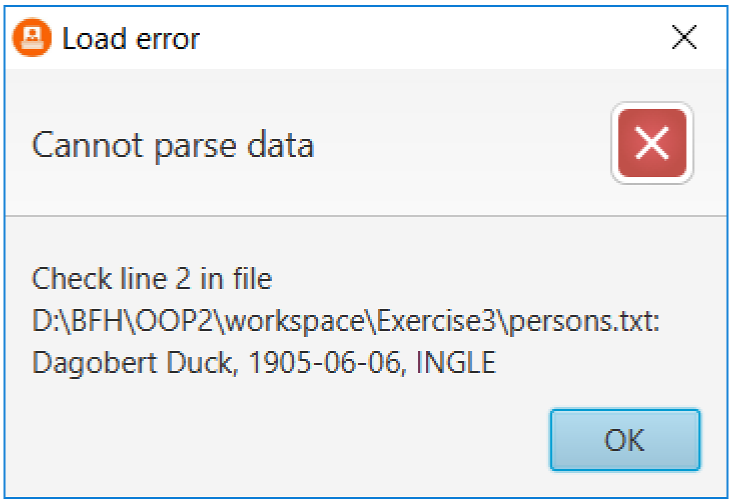
So far, we considered file related exceptions. Once you have successfully opened your text file, you must read it line by line, parse each line and create a Person object. You have already done this in exercises 2.

What happens, if you have lines in your text file, which do not comply with the expected format? Parsing will fail and a *RuntimeException* will be thrown.

Analyse your parser code and write a *JavaDoc* comment that lists all the possible runtime exceptions together with a description of the circumstances, that lead to the exception being thrown. Can you prevent some of these exceptions being thrown by checking some conditions first? If yes, do so!

5. Runtime Exceptions during Parsing of Person Data

The remaining runtime exceptions, possibly thrown by your parser code, shall be catched by your text file DAO and chained to a specific *PersistenceException*. Therefore, create a sub-class of *PersistenceException*. To provide the user with additional info about the problem, the class shall contain the file name of the parsed file, the line number and the line that causes the parser to fail. Declare the appropriate constructor and getter methods. If a parse error occurs, your application shall open an Alert dialog displaying this information.


Load error

Cannot parse data

Check line 2 in file
D:\BFH\OOP2\workspace\Exercise3\persons.txt:
Dagobert Duck, 1905-06-06, INGLE