HSLU Hochschule Luzern

Objektorientierte Programmierung (OOP)

Übung SW09: Fehlerhandling und Logging

Themen: Fehlerhandling und Exceptions (O11), Testing und Logging (E03).
Zeitbedarf: ca. 240min.

Roland Gisler, Version 1.7.3 (FS25)

1 Exceptionhandling Grundlagen

1.1 Lernziele

- Einfache Ausnahmebehandlung mit `try`/`catch`-Block.
- Analyse eines einfachen, vorhandenen Programmes.
- Erweiterung der Funktionalität.

1.2 Grundlagen

Gegeben ist das folgende Codegerüst für eine `main()`-Methode:

```
public static void main(String[] args) {
    String input;
    Scanner scanner = new Scanner(System.in);
    do {
        System.out.println("Bitte Temperatur eingeben (oder 'exit' zum Beenden): ");
        input = scanner.next();
        float value = Float.valueOf(input);
    } while (!"exit".equals(input));
    System.out.println("Programm beendet.");
}
```

1.3 Aufgaben

a.) Studieren Sie den Code und versuchen Sie grob zu verstehen was passiert. Greifen Sie dafür auf die JavaDoc zurück.

b.) Kopieren Sie den Code in eine eigene Klasse und testen Sie ihn manuell aus.
Was stellen Sie fest? Entspricht das Verhalten Ihren Erwartungen von a)?

c.) Eine der verwendeten Methoden wirft offenbar in bestimmten Situationen eine Exception. Welche Methode und welche Exception ist das? Studieren Sie die JavaDoc sowohl dieser Methode als auch der geworfenen Exception. Ist es eine «checked exception»?

d.) Überlegen Sie sich wie das Fehlerhandling aussehen muss, damit das Programm sich korrekt verhält und nicht mit einem Stacktrace abgebrochen wird!

e.) Implementieren Sie die folgenden Anforderungen: Wenn eine gültige Zahl (bzw. Temperatur) eingegeben wird, soll diese Zahl auf der Konsole ausgegeben werden. Wenn nein, soll ebenfalls auf der Konsole ein Hinweis ausgegeben, das Programm aber weiterlaufen. Und mit «exit» soll das Programm fehlerfrei beendet werden.

f.) Haben Sie daran gedacht, dass wir hier unsere `Temperatur`-Klasse zu Hilfe nehmen könnten? Gut! Gehen Sie zur nächsten Aufgabe.

2 Exceptionhandling für die Klasse `Temperatur`

2.1 Lernziele

- Fehlerbehandlung durch werfen einer Exception.
- Testen des Fehlerhandlings mit Unit-Tests.
- Konzept einer Immutable Klasse.

2.2 Grundlagen

Einmal mehr wollen wir unsere **`Temperatur`**-Klasse im Rahmen dieser Aufgabe weiter verbessern und überarbeiten. Dabei geht es um zwei Dinge:

- Bisher konnten wir **`Temperatur`**-Objekte mittels eines Konstruktors (für Celsius) erzeugen und die Temperatur danach mit Setter-Methoden (sowohl für Celsius als auch Kelvin) verändern. Eigentlich wäre es doch schön, wenn wir direkt eine Temperatur in der gewünschten Einheit erzeugen könnten, oder?
- Was wir bisher auch völlig ausser Acht gelassen haben, ist dass wir beliebige Temperaturwerte setzen können. Das ist physikalisch falsch: Es gibt keine Temperaturen die tiefer als der absolute Nullpunkt von **`-273.15`**°C (bzw. **`0`**K) sind. Das müssen wir bei unserer **`Temperatur`**-Klasse somit auch noch berücksichtigen!

2.3 Aufgaben

Teil 1:

a.) Überlegen Sie: Gibt es eine elegante Möglichkeit, wie wir zwei Konstruktoren anbieten könnten, welche je für die Einheiten Celsius und Kelvin **`Temperatur`**-Objekte erstellen können? Was haben Sie für Varianten? (Der Name ist ja vorgegeben!)

b.) Vermutlich sind Sie auf die Idee gekommen die gewünschte Einheit per zusätzlichem Parameter zu übergeben. Aber was verwenden Sie dafür für einen Typ? Wenn Sie diesen Weg weiterverfolgen wollen (probieren Sie es ruhig aus, es ist eine gute Übung!), denken Sie unbedingt an die Möglichkeit der Enumerationen!

c.) Es gibt noch eine weitere Möglichkeit, die uns sogar flexible Methodennamen erlaubt: Wir können Objekte einer Klasse auch über statische Methode erzeugen! Diese Technik bezeichnet man als sogenannte Factory-Methoden[1] (weil sie wie eine Fabrik «auf Auftrag» Objekte erzeugen).
Erstellen Sie eine statische Methode mit folgendem Methodenkopf:
`public static Temperatur createFromCelsius(final float celsius)`
Tipp: In dieser Methode rufen Sie einen vorhandenen Konstruktor (und ggf. andere) Methoden auf.

d.) Es sollte ein leichtes sein, nun noch eine weitere Methode **`createFromKelvin(…)`** anzubieten. Aber Vorsicht: Produzieren Sie dabei möglichst wenig Coderedundanzen!

e.) Tatsächlich können wir nun sogar gänzlich verhindern, dass jemand direkt über den Konstruktor ein Objekt erzeugt, und damit sogar erzwingen, dass die Factory-Methoden verwendet werden: Setzen Sie die Sichtbarkeit aller Konstruktoren auf **`private`**! Vielleicht sind Sie überrascht, dass das nicht nur funktioniert, sondern in manchen Fällen (wie hier) sogar sehr sinnvoll ist.

[1] Factory-Methoden sind ein Entwurfsmuster der GoF. Mehr dazu im Modul VSK im 3. oder 4. Semester.

Teil 2:

f.) Wir wollen verhindern, dass wir unerlaubte Temperaturwerte definieren können. Ein typischer Fall für eine Ausnahme! Aber welchen Exceptiontyp verwenden wir dafür?
Tipp: Schauen Sie sich in der JavaDoc die Spezialisierungen von `RuntimeException` an!

g.) Wenn wir einen ungültigen Temperaturwerte verwenden, übergeben wir einer Methode ein ungültiges Argument (aktueller Parameter): Somit können wir die bestehende (unchecked) Exception vom Typ `IllegalArgumentException` verwenden! Studieren Sie die JavaDoc dieser Exception!

h.) Erweitern Sie Ihre `Temperatur`-Klasse entsprechend, dass bei unerlaubten Temperaturwerten eine `IllegalArgumentException` geworfen wird. Setzten Sie dabei eine aussagekräftige, aber kurze Message!

i.) Funktionieren alle bisherigen Unit-Tests noch oder haben Sie darin (absichtlich oder versehentlich?) auch unerlaubte Temperaturwerte benutzt? Ergänzen und korrigieren Sie ihre Testfälle!

j.) Sie finden im Input `E03_IP_ExceptionTestingLogging` unter dem Abschnitt «Testen des Exceptionhandlings» eine kurze Anleitung zum Testen von Exceptions mit JUnit. Ergänzen Sie entsprechende Testfälle für ungültige Temperaturwerte.

Wichtig: Prüfen Sie bei der Implementation der Testfälle unbedingt, ob diese auch tatsächlich failen wenn z.B. der falsche Exceptiontyp (oder mit einer falscher Message) geworfen wird.

k.) Macht es in dieser Klasse eigentlich noch Sinn, dass wir Temperaturwerte nachträglich mit Setter-Methoden verändern können? Ist es nicht einfacher, einfach ein neues Temperatur-Objekt zu erzeugen? Entfernen Sie einfach mal alle Setter-Methoden!

Wieder spannend: Wie gut haben Sie getestet, und wie viele Testmethoden werden dadurch jetzt obsolet? Auf jeden Fall wird die Klasse wieder ein gutes Stück kleiner und einfacher, oder?
Haben Sie daran gedacht, dass das enthaltene Attribut jetzt `final` deklariert werden kann (bzw. sollte?)

Hinweis: Damit haben wir eine Klasse für so genannte «immutable objects» erstellt. Das Potential dieses Designentscheides wird sich uns aber erst später erschliessen, siehe dazu z.B. unter https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html.

Optionale Teilaufgaben (voneinander unabhängig, Auswahl möglich):

l.) Fachlich in diesem Beispiel **nicht** sinnvoll, können Sie trotzdem (zu reinen Übungszwecken) schon jetzt ihre erste **eigene Exception** implementieren!
Spezialisieren Sie dazu von `Exception` (somit definieren Sie eine «checked exception») und geben Sie ihr einen sinnvollen Namen. Was müssen Sie im Code alles ändern, wenn Sie nun Ihre eigene Exception verwenden?

m.) Das Exeption-Handling (und anderes) lässt sich noch ein bisschen eleganter und einfacher testen, wenn Sie dazu die Library AssertJ zu Hilfe nehmen, welche im Projekt bereits integriert ist. Studieren Sie die Website, sie enthält ein sehr gutes «Quick start»: https://assertj.github.io/doc/ und probieren Sie es aus!

3 Logging mit SLF4J und LogBack

3.1 Lernziele

- Verwendung eines Logging-Frameworks.
- Erste Erfahrung im gezielten, sinnvollen Logging sammeln.
- Potential der Logging-Technologie verstehen.

3.2 Grundlagen

Im zur Verfügung gestellten Projekttemplate ist «Logback over SLF4J» bereits integriert und vorkonfiguriert. Sie können es somit mit minimalem Aufwand für all Ihre Beispiele / Programme nutzen und damit die eigentlich hässlichen und unflexiblen `System.out.println(…)` konsequent durch professionelles Logging ersetzen.

Im Input `E03_IP_ExceptionTestingLogging` finden Sie im letzten Abschnitt «Logging Technologie» eine kurze Anleitung zur Verwendung von SLF4J und Logback. Zusätzliche Informationen finden Sie unter https://slf4j.org/ und https://logback.qos.ch/.

3.3 Aufgaben

a.) Verwenden Sie z.B. das Beispiel von Aufgabe 1 und ersetzen Sie die enthaltenen Hilfs- und Debug-`System.out.println(…)`-Statements durch Logging. Beim «holen» des Loggers achten Sie darauf, dass Sie jeweils Ihre **aktuelle** Klasse als Parameter übergeben!

Achtung: Das Beispiel ist eine Konsolenanwendung, einige Dinge sollen/wollen wir tatsächlich explizit über die Konsole ausgeben! ➔ Damit wird auch klar, dass wir die Konsole häufig unbewusst für mindestens zwei verschiedene Dinge verwendet haben!

b.) Probieren Sie einfach mal die verschiedenen Levels aus! In der Praxis ist es wichtig (und eine Herausforderung) für jeden Logeintrag den korrekten, sinnvollen Level zu verwenden!

c.) Loggen Sie auftretende Exceptions mit dem `Error`-Level und geben Sie die Exception (`Throwable`) an das Log mit!

d.) Werfen Sie einen Blick in die Konfigurationsdatei von Logback, welche im Projekt unter dem Pfad `./src/main/resources/logback.xml` abgelegt ist[2]. Beachten Sie, dass Sie im Element `configuration → root` ein Attribut `level` finden, mit welchem Sie festlegen können, bis und mit welchem Level die Ausgabe der Logs stattfindet!

Optional:

e.) Fühlen Sie sich frei zu Experimentieren und auszuprobieren. Sie können z.B. das Format verändern. Sie können weitere Appender (Log-Senken) definieren, z.B. in eine Datei oder für einen Cloud-Dienst. Und vieles weiteres mehr!

Empfehlung: Sie müssen Sich hier und jetzt nicht mit allen Details des Logging-Systems auseinandersetzen. Aber verwenden Sie es ab sofort mindestens als «Black-Box», indem Sie im Sinne einer Professionalisierung statt `System.out.println(…)` wo immer sinnvoll die Logging-Technologie nutzen, um Dinge auszugeben, die rein der Kontrolle oder zum Nachvollziehen des Ablaufes (Tracing) dienen.

[2] Wenn Sie für die Ausführung der Testfälle eine spezielle Logger-Konfiguration nutzen wollen, können Sie unter `/src/test/resources` eine weitere Konfiguration ablegen, welche von Apache Maven ausschliesslich während der Testausführung genutzt wird.

4 Optionale Repetitionsaufgabe: Raumverwaltung – Teil 1 von 5

4.1 Lernziele

- Repetition bereit behandelter Themen zur Festigung und Vertiefung.

4.2 Grundlagen

Beginnend mit dieser Übungseinheit erhalten Sie jede Woche eine optionale Zusatzaufgaben zur Repetition aller bisher behandelter Themen. Diese Zusatzaufgaben sind unabhängig von den restlichen Aufgaben dieser Woche. Sie bauen aber ihrerseits aufeinander auf.

Optionale Aufgaben sind **nicht** Bestandteil der Besprechungen.

4.3 Aufgaben

a.) Erstellen Sie eine nicht spezialisierbare Klasse `Raum` mit je einem Attribut für die Raumnummer (Ganzzahl) und der maximalen Platzanzahl (Kapazität). Nur Klassen im selben Package `ch.hslu.oop.rv` sollen Raumobjekte erzeugen können. Die Raumnummer und die Platzanzahl sollen direkt über den Konstruktor einmalig (immutable) gesetzt werden. Beachten Sie die Datenkapselung.

b.) Testen Sie mit einem Unit-Test, ob der Konstruktor die Attribute korrekt setzt.

c.) Prüfen Sie im Konstruktor die Parameter, so dass nur Raumnummern im Bereich von `100` bis `999` und nur Kapazitäten grösser als zwei Plätze möglich sind. Werfen Sie im Fehlerfall eine sinnvolle Exception mit individueller Meldung. Testen Sie mit drei Unit-Tests alle (Grenz-)Fälle.

d.) Implementieren Sie den equals-Contract auf der Klasse `Raum`. Zwei Räume sollen gleich sein, wenn die Raumnummer identisch ist.