Modellierung und Programmierung 1

Übung 8

Stefan Preußner

11./ 12. Januar 2021

Ausnahmen

Exceptions und Errors

- In vielen alten Programmiersprachen ist es üblich, Funktionen -1 oder 0 zurückgeben zu lassen, wenn ein Fehler aufgetreten ist
- Probleme
  - Keine Rückgabe eines Fehlercodes möglich, wenn z.B. jede mögliche Ganzzahl ein regulärer Rückgabewert einer Funktion ist
  - Mehrere mögliche Fehler erfordern ggf. verschiedene Fehlercodes
  - Analyse der Fehlerursache schwierig

Exceptions und Errors

- In Java können beim Auftreten von Fehlern **Exceptions** und **Errors** ausgelöst werden
- `Exception` und `Error` sind Unterklassen der Klasse **Throwable**
- `Exceptions` sind vor allem für Fehler gedacht, die innerhalb des Programms behandelt werden können
- `Errors` sind vor allem für schwerwiegende Fehler gedacht, die nicht behandelt werden sollten und in der Regel einen Programmabbruch zur Folge haben

try und catch

- Mit den Schlüsselwörtern **try** und **catch** können Fehler abgefangen und behandelt werden
- Innerhalb des `try`-Blocks steht der Code, welcher eine Exception auslösen kann:

```
try
{
    // Code, welche eine oder mehrere
    // Exceptions auslösen kann
}
```

try und catch

- Nach `catch` folgt in runden Klammern der Name des Fehlers (eine von `Throwable` abgeleitete Klasse), der behandelt werden soll und in anschließend in geschweiften Klammern der Code zur Fehlerbehandlung:

```
catch (ExceptionTyp fehlervariable)
{
    // Code, welcher die Fehlerbehandlung durchführt
    // Der Fehler ist in fehlervariable gespeichert
    // und kann analysiert werden
}
```

try und catch

- Die `catch`-Anweisung gilt automatisch für alle Unterklassen der angegebenen Exception
  - Tritt eine `Exception` auf, die nicht explizit behandelt wird, so wird automatisch überprüft, ob ihre Oberklasse behandelt wird
  - Bspw. können mit **catch (Exception e)** alle in Java vordefinierten Exceptions abgefangen werden
- Auf eine `try`-Anweisung können beliebig viele `catch`-Anweisungen folgen
  - Der erste `catch`-Block mit einer passenden Fehlerklasse wird ausgeführt, `catch (Exception e)` sollte (wenn überhaupt) also immer als letztes im Code stehen

try und catch

```
public int dividiereGanzzahlig(int x, int y)  {
    return x/y;    // wirft bei y == 0 eine ArithmeticException
}

for (int k = 2; k >= -2; k--)
{
    try {
        dividiereGanzzahlig(9, k);
    }
    catch (ArithmeticException e) {
        System.out.println("Fehler bei k = " + k + ": " + e.getMessage());
    }
}
```

Abfangen mehrerer Exceptions

```
public class ZahlenArray {
    public int[] daten;
}

public static void leseDaten(ZahlenArray arr, int index) {
    try {
        System.out.println(arr.daten[index]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Index ist ungueltig!");
    }
    catch (NullPointerException e) {
        System.out.println("Das ZahlenArray darf nicht null sein!");
    }
}
```

Multi-catch

- Mehrere unterschiedliche Fehler können zusammen abgefangen und gleich behandelt werden:

```
public class ZahlenArray {
    public int[] daten;
}

public static void leseDaten(ZahlenArray arr, int index) {
    try {
        System.out.println(arr.daten[index]);
    }
    catch (ArrayIndexOutOfBoundsException | NullPointerException e) {
        System.out.println("Fehler!");
    }
}
```

finally

- Nach dem letzten `catch`-Block kann optional ein **finally**-Block folgen
- Code im `finally`-Block wird immer ausgeführt, unabhängig davon, ob ein Fehler aufgetreten ist
- Beispiel für einen sinnvollen Einsatz:
  - Eine Datei wird geöffnet, ausgelesen und der Inhalt verarbeitet
  - Die Datei kann beschädigt sein, in diesem Fall kann der Inhalt nicht verarbeitet werden
  - In jedem Fall sollte die Datei nach dem Auslesen wieder geschlossen werden
  - Das Schließen der Datei erfolgt im `finally`-Block

finally

```
public class ZahlenArray {  public int[] daten;  }

public static void leseDaten(ZahlenArray arr, int index) {
    try {
        System.out.println(arr.daten[index]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Index ist ungueltig!");
    }
    catch (NullPointerException e) {
        System.out.println("Das ZahlenArray darf nicht null sein!");
    }
    finally {
        System.out.println("Die Argumente des Funktionsaufrufs
        waren arr=" + arr + " und index=" + index);
    }
}
```

throws

- Eine Exception kann, statt sie zu behandeln, an die aufrufende Funktion weitergeleitet werden
- Dies geschieht durch das Schlüsselwort **throws** im Methodenkopf:

```
public void foo()
    throws ArrayIndexOutOfBoundsException, NullPointerException {
    // Code, der unter Umstaenden eine der oben angegebenen
    // Exceptions ausloest
}
```

- Die Fehlerbehandlung (ignorieren, `catch`, Weiterleitung) wird dann der aufrufenden Funktion überlassen

throws - Beispiel

```
public class ZahlenArray {
    public int[] daten;
}

public static void leseDaten(ZahlenArray arr, int index)
    throws ArrayIndexOutOfBoundsException, NullPointerException {
    System.out.println(arr.daten[index]);
}
```

Geprüfte und ungeprüfte Exceptions

- Java unterscheidet zwischen geprüften (*checked*) und ungeprüften (*unchecked*) Exceptions
- Ungeprüfte Exceptions müssen nicht mit `try` und `catch` abgefangen werden (daher der Name: der Compiler prüft bei der Compilierung nicht, ob die Exception behandelt wird)
  - Tritt eine solche Exception außerhalb eines `try`-Blocks auf, wird das Programm beendet
  - Vom Konzept her deuten auftretende ungeprüfte Exceptions auf Fehler im Programmcode hin und sollten anderweitig vermieden werden

Geprüfte und ungeprüfte Exceptions

- Geprüfte Exceptions müssen immer behandelt werden
  - Löst eine Methode eine geprüfte Exception aus, so muss dies im Methodenkopf mit dem Schlüsselwort `throws` angegeben werden
  - Geprüfte Exceptions sind vor allem *extern* verursachte Fehler, die sich schwer oder nicht durch programminterne Überprüfungen vermeiden lassen (Ein-/Ausgabefehler, Fehler beim Datenbankzugriff, *Interrupts*, ...))

Die Klasse RuntimeException

- **RuntimeException** ist eine Oberklasse, von der alle **ungeprüften** Exceptions abgeleitet werden
- Alle von RuntimeException abgeleiteten Klassen sind automatisch ungeprüfte Exceptions, alle anderen sind automatisch geprüfte Exceptions
- Beim Schreiben von eigenen Fehlerklassen muss abgewägt werden, ob von RuntimeException oder von Exception abgeleitet wird

Auslösen von Exceptions

- Eine Exception wird mit dem Schlüsselwort **throw** geworfen
- Syntax:

```
throw new NullPointerException();
throw new ArrayIndexOutOfBoundsException("Der Index ist ungueltig!");
```

- Alle von Throwable abgeleiteten Klassen haben vier wichtige Konstruktoren:
  - Throwable() - Keine Fehlermeldung
  - Throwable(String s) - Mit Fehlermeldung
  - Throwable(Throwable ursache) - Vor allem gedacht zum Weiterleiten von Fehlern an aufrufende Funktionen
  - Throwable(String s, Throwable ursache)

Eigene Fehlerklassen

- Eigene Fehlerklassen können von beliebigen Throwable-Klassen abgeleitet werden (wobei am häufigsten von RuntimeException oder Exception geerbt wird):

```
public class ZahlZuGrossException extends Exception {

    public ZahlZuGrossException(String fehlermeldung) {
        super(fehlermeldung);
    }

}
```

- Wichtig: Aufruf von `super` mit den geeigneten Parametern

Stacktrace

- Auf dem Stack werden (u.a.) die Rücksprungadressen der Funktionen abgelegt, die eine neue Funktion aufrufen
- Der **Stacktrace** ist eine lesbare Ausgabe des Stacks
- Durch den Stacktrace kann bei einem Fehler festgestellt werden, welche Kette von Funktionsaufrufen zum Fehler geführt hat

Stacktrace - Beispiel

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ZahlenArray.qux(ZahlenArray.java:16)
at ZahlenArray.bar(ZahlenArray.java:21)
at ZahlenArray.foo(ZahlenArray.java:26)
at Main.main(Main.java:25)
```

- Funktion `Main.main()` ruft in Zeile 25 der `Main.java` die ...
- Funktion `Zahlenarray.foo()` auf, welche in Zeile 26 der `Zahlenarray.java` die ...
- Funktion `Zahlenarray.bar()` aufruft, welche in Zeile 21 der `Zahlenarray.java` die ...
- Funktion `Zahlenarray.qux()` aufruft, welche in Zeile 16 der `Zahlenarray.java` eine `ArithmeticException` auslöst

Stacktrace

- Zur Ermittlung des Stacktraces gibt es zwei Methoden:
- Throwable.printStackTrace() gibt den Stacktrace auf der Konsole aus
  - Die Methode ist überladen, um die Ausgabe z.B. in eine Datei umleiten zu können
- Throwable.getStackTrace() gibt den Stacktrace als `StackTraceElement[]` zurück

Häufige Exceptions

- ArithmeticException
  - Ursache: mathematischer Fehler, z.B. Ganzzahldivision durch 0 oder Logarithmieren einer negativen Zahl
  - Vermeidung: Test auf $!= 0$, $>= 0$ usw.
- IndexOutOfBoundsException, ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException
  - Ursache: ungültiger Index beim Zugriff auf ein Array oder einen String
  - Vermeidung: Test, dass der Index nicht negativ und kleiner als die Länge der Datenstruktur ist

Häufige Exceptions

- ClassCastException
  - Ursache: Objekt wird auf eine Subklasse gecastet, von der es keine Instanz ist
  - Vermeidung: Test mit instanceof und andere Tests auf Klassenzugehoerigkeit; ggf. Überladen von Funktionen, um Oberklassen als Parameter zu vermeiden
- NullPointerException
  - Ursache: Programm erwartet ein Objekt, erhält aber einen Null-Zeiger
  - Vermeidung: Test auf null

Häufige Exceptions

- IllegalArgumentException, NumberFormatException, IllegalFormatException
  - Ursache: einer Funktion wurde ein ungültiges Argument übergeben (z.B. `Integer.parseInt("ABC")`)
  - Vermeidung: kaum möglich, wenn z.B. im Programm in irgendeiner Form Nutzereingaben möglich sind
  - Solche Exceptions sollten möglichst immer mit `try/catch` abgefangen werden

Häufige Exceptions und Errors

- IOException
  - Ursache: Oberklasse aller Ein- und Ausgabefehler
  - Vermeidung: IOException sind geprüfte Exceptions, eine Vermeidung ist nicht vorgesehen, stattdessen sollen diese Exceptions behandelt werden
- StackOverflowError
  - Ursache: maximale Größe des Stacks wird überschritten, i.d.R. durch eine zu hohe Rekursionstiefe
  - Vermeidung: Änderungen am Programmablauf/Algorithmus
  - Ein StackOverflowError ist ein Error, keine Exception, und sollte daher nicht behandelt werden

Ein- und Ausgabe

Ein- und Ausgabe

- Arten von Ein- und Ausgaben:
  - Abfragen von Nutzereingaben in der Konsole
  - Einlesen von Dateien bzw. Schreiben in Dateien
  - Netzwerkdatenströme
  - Daten von Peripheriegeräten
  - usw.
- Wir beschäftigen uns ausschließlich mit Dateien

`java.io` und `java.nio`

- Java stellt zwei Pakete für die Ausgabe in bzw. das Lesen von Dateien zur Verfügung: `java.io` und `java.nio`

- `java.io`:
  - Datenstromorientiert (Datenstrom = Stream)
  - Daten werden ungepuffert und sequentiell eingelesen, d.h. ein Zurückspringen im Datenstrom ist nicht möglich (außer durch eine eigene Implementierung eines Puffers)
  - Dateien werden beim Lesen bzw. Schreiben blockiert, d.h. es kann immer nur ein Thread auf eine Datei zugreifen

`java.io` und `java.nio`

- `java.nio`:
  - Pufferorientiert
  - Daten werden in einen Puffer gelesen, in dem sich frei bewegt werden kann (ein Zurückspringen ist also möglich)
  - Dateien werden beim Lesen bzw. Schreiben nicht blockiert (`nio` = non-blocking I/O), d.h. es können gleichzeitig und parallel Daten gelesen und geschrieben werden

Zeichen- und Byte-basierte Ein- und Ausgabe

- Daten können im Allgemeinen in zwei Kategorien aufgeteilt werden: Zeichenfolgen und Bytefolgen
- Zeichenfolgen werden immer dann gespeichert, wenn Daten menschlich lesbar sein sollen
  - reiner Text, HTML, XML, Base64, ...
- Bytefolgen werden dann verwendet, wenn die exakte Reihenfolge von Bits wichtig ist
  - praktisch alle Bild-, Musik- und Videoformate, Speicherung von Rohdaten, ...

Zeichen- und Byte-basierte Ein- und Ausgabe

- In `java.io` existieren vier Basisklassen für die Ein- und Ausgabe:

	Für Bytefolgen	**Für Zeichenfolgen**
Eingabe	`InputStream`	`Reader`
Ausgabe	`OutputStream`	`Writer`

- Alle hiervon abgeleiteten Klassen tragen den Namen dieser Oberklassen jeweils am Ende ihres eigenen Namens
  - Ein `StringBufferInputStream` ist also - trotz des `String` im Namen - für die Eingabe von Bytefolgen gedacht

- Für die Ein- und Ausgabe aus/in Dateien gibt es die folgenden vier Basisklassen:

	Für Bytefolgen	**Für Zeichenfolgen**
Eingabe	`FileInputStream`	`FileReader`
Ausgabe	`FileOutputStream`	`FileWriter`

FileReader

- Basisklasse zum Lesen von Zeichenfolgen aus Dateien
- Dem Konstruktor kann entweder ein `File`-Objekt

```
File f = new File("Hallo.txt");
FileReader r = new FileReader(f);
```

oder ein String

```
FileReader r = new FileReader("Hallo.txt");
```

oder ein `FileDescriptor`-Objekt übergeben werden

FileReader

- Das Lesen von Daten erfolgt mit der Methode `read`
  - `read()` liest einen einzelnen Buchstaben und gibt ihn als `int` zurück. Gibt -1 zurück, wenn das Ende der Datei erreicht wurde.
  - `read(char[] puffer)` liest Buchstaben in einen `char`-Array ein. Gibt die Anzahl der eingelesenen Zeichen zurück oder -1, wenn das Ende der Datei erreicht wurde.
- $\rightarrow$ FileReader sind für das zeichenweise Einlesen ausgelegt und erfordern eine relativ aufwendige Verarbeitung der eingelesenen Zeichen

BufferedReader

- Übernimmt im Konstruktor einen anderen `Reader` und benutzt dann diesen, um Daten gepuffert einzulesen
- Wichtigste Funktion: `readLine()` - liest Dateien zeilenweise ein

```
FileReader fr = new FileReader("Hallo.txt");
BufferedReader br = new BufferedReader(fr);
String zeile = br.readLine();
```

- `readLine()` gibt `null` zurück, wenn das Dateiende erreicht wurde

FileInputStream

- Klasse zum Lesen von Bytefolgen aus Dateien
- Dem Konstruktor kann (äquivalent zu `FileReader`) ein `File`-Objekt oder ein Dateiname übergeben werden
- Die wichtigsten Funktionen:
  - `int read()` - gibt das nächste Byte in der Datei zurück. Gibt beim Erreichen des Dateiendes -1 zurück.
  - `int read(byte[] b)` - liest `b.length` viele Bytes aus der Datei und speichert sie in `b`. Gibt die Anzahl der gelesenen Bytes zurück.
  - `long skip(long n)` - überspringt `n` viele Bytes in der Datei

FileWriter

- `FileWriter` ermöglicht das Schreiben von Zeichenfolgen in Dateien
- Dem Konstruktor kann (äquivalent zu `FileReader`) entweder ein `File`-Objekt, ein String oder ein `FileDescriptor`-Objekt übergeben werden
- Daneben existieren die Konstruktoren `FileWriter(File f, boolean append)` und `FileWriter(String s, boolean append)`, mit denen durch Setzen von `append` auf `true` eine Datei im Anhängemodus geöffnet werden kann

FileWriter

- `FileWriter` stellt folgende (von `Writer` geerbte) Methoden zur Verfügung:
  - `write(char c)` schreibt ein einzelnes Zeichen
  - `write(char[] carr)` schreibt ein `char`-Array
  - `write(String s)` schreibt einen String
  - `append(char c)`
  - `append(CharSequence s)`

FileWriter und BufferedWriter

- Da das Schreiben sehr vieler einzelner Zeichen sehr ineffizient ist, sollte ein `FileWriter` innerhalb einer anderen `Writer`-Klasse wie `BufferedWriter` verwendet werden:

```
try {
    FileWriter fw = new FileWriter("Hallo.txt", true);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write("MuP ist toll!");
    bw.close();
}
catch (IOException e) {
    System.out.println("Fehler!");
}
```

close()

- Erinnerung: alle Klassen in `java.io` blockieren Dateien beim Öffnen
- Dateien sollten daher immer mit der Funktion `close()` geschlossen werden, sobald mit ihnen nicht mehr gearbeitet wird
- Dies gilt insbesondere beim Schreiben in Dateien[1]

[1] `close()` ruft automatisch die Methode `flush()` auf. `flush()` leert evtl. vorhandene Ausgabepuffer und überträgt alle noch zu schreibenden Daten (z.B. an das Betriebssystem).

try mit Ressourcen

- Nach einem `try` können, in **runden Klammern**, **Ressourcen** wie Reader und Writer (also effektiv Datenströme) angegeben werden:

```
try (BufferedReader br = new BufferedReader(new FileReader("a.txt")))
{
  // Code, welcher den Reader br nutzen kann
}
```

- Am Ende des `try`-Anweisungsblocks oder bei Fehlern beim Lesen/Schreiben werden alle Ressourcen, die `java.lang.AutoCloseable` implementieren, automatisch geschlossen und freigegeben

try mit Ressourcen

```
public static ArrayList<String> leseZeilenweise(String datei)
    throws IOException
{
    ArrayList<String> alleZeilen = new ArrayList<String>();
    try (BufferedReader br = new BufferedReader(new FileReader(datei)))
    {
        while (true)
        {
            // lese zeilenweise bis zum Ende der Datei
            // Abbruch der Endlosschleife mittels break, falls die
            // eingelesene Zeile gleich null ist
        }
    }
    return alleZeilen;
}
```

java.nio.file

- `java.nio.file` ist ein Paket, welches Klassen und Schnittstellen für den Zugriff auf Dateien, Dateiattribute und Dateisysteme zur Verfügung stellt
- `java.nio.file.Files` ist eine Klasse, welche zahlreiche statische Funktionen für die Ein- und Ausgabe sowie für die Verwaltung von Dateien und Ordnern zur Verfügung stellt
- `java.nio.file.Paths` stellt statische Methoden zur Erzeugung von `Path`-Objekten (Datei- bzw. Ordnerpfaden) zur Verfügung

java.nio.file

- `public static Path get(String pfad)` in der Klasse `Paths` erzeugt ein `Path`-Objekt aus dem angegebenen Dateipfad
- `public static List<String> readAllLines(Path p)` in der Klasse `Files` liest alle Zeilen einer zeichenbasierten Datei ein und gibt sie als Liste von Strings zurück
- `public static byte[] readAllBytes(Path pfad)` in der Klasse `Files` liest eine bytebasierte Datei vollständig aus und gibt den Dateiinhalt als Byte-Array zurück
- Achtung: die beiden Lesemethoden laden den kompletten Dateiinhalt in den Speicher, was bei großen Dateien problematisch sein kann

java.nio

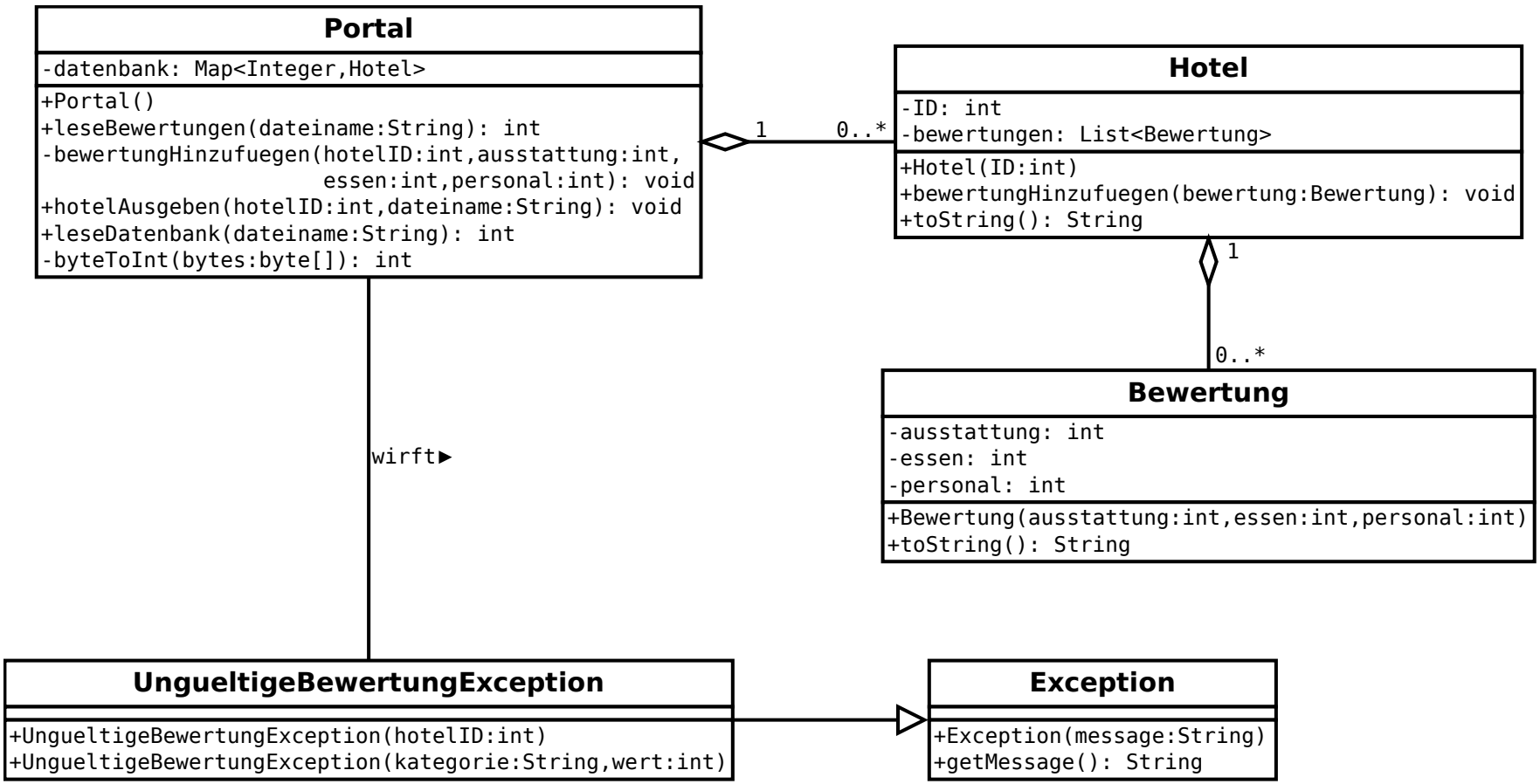
```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public static void main(String[] args)) {
    try {
        Path pfad = Paths.get("Hallo.txt");
        List<String> zeilen = Files.readAllLines(pfad);
        System.out.println(zeilen);
    }
    catch (IOException e) {
        System.out.println("Fehler!");
    }
}
```

Programmierübung

Ein Bewertungsportal für Hotels möchte einen Teil seiner Datenbankverwaltung neu organisieren. Sie haben die Aufgabe bekommen, sich insbesondere um das Einlesen der Datenbank zu kümmern. Nach der Modellierungsphase planen Sie, das Programm auf der nachfolgenden Folie zu realisieren.

Programmierübung - Hotelbewertungen



Implementieren Sie die Methode `bewertungHinzufuegen(int hotelID, int ausstattung,int essen, int personal)` der Klasse Portal, welche eine neue Bewertung erzeugt und zu dem entsprechenden Hotel hinzufügt.
Die Methode soll eine Ausnahme werfen, wenn entweder die ID des Hotels nicht in der Datenbank enthalten ist oder wenn die Punktzahl in einer der drei Kategorien außerhalb des gültigen Wertebereichs von 0 bis 10 liegt. Erstellen Sie hierzu eine Klasse `UngueltigeBewertungException`, welche von der Java-Klasse Exception erben soll. Bei einer ungültigen ID soll die Fehlermeldung (der von `getMessage()` zurückgegebene String) die Form

```
ID 128513 nicht gefunden
```

haben. Die Fehlernachricht bei einer ungültigen Teilbewertung in einer der drei Kategorien soll die Form

```
Ungültige Bewertung (-10) in der Kategorie Essen
```

haben.

Implementieren Sie die Methode `leseBewertungen(String dateiname)` der Klasse Portal, welche die Hotelbewertungen aus der Bewertungsdatei `dateiname` einliest. Tritt beim Öffnen oder Einlesen der Datei ein Fehler auf, dann soll eine aussagekräftige Meldung auf der Konsole ausgegeben werden.
Die Datei enthält je Zeile genau eine Bewertung im folgendem Format:

`HotelID;WertungAusstattung;WertungEssen;WertungPersonal`

Hierbei ist `HotelID` die ID des bewerteten Hotels; `WertungAusstattung`, `WertungEssen` und `WertungPersonal` sind die vergebenen Punkte in den Kategorien Ausstattung, Essen und Personal. Fügen Sie die Bewertungen unter Verwendung der Methode `bewertungHinzufuegen` zum jeweiligen Hotel hinzu.

Sollte eine Bewertung ungültig sein, dann soll eine Fehlermeldung in dem folgenden Format auf der Konsole ausgegeben werden:

`Fehler in Zeile <Zeilennummer> von <Dateiname>: <Fehlermeldung>`

Beispiel:

`Fehler in Zeile 131 von Bewertungen.txt: Ungültige Bewertung (11) in der Kategorie Ausstattung`

Das Einlesen der Bewertungsdatei soll bei einer ungültigen Bewertung nach Ausgabe der Fehlermeldung fortgesetzt werden.
Geben Sie die Gesamtzahl der eingelesenen Bewertungen (einschließlich der ungültigen) zurück.

Hinweis: Sie können annehmen, dass die Formatierung sämtlicher Zeilen der Bewertungsdatei korrekt ist.

Implementieren Sie die Methode `hotelAusgeben(int hotelID, String dateiname)` der Klasse Portal, welche für das Hotel mit der ID `hotelID` den von der `toString()`-Methode zurückgegebenen String in die Datei `dateiname` schreibt. Tritt beim Erstellen/Öffnen oder Schreiben ein Fehler auf, dann soll eine aussagekräftige Fehlermeldung auf der Konsole ausgegeben werden.

Hinweis: Sie können annehmen, dass die Methode nur für gültige IDs aufgerufen wird.

Führen Sie in der `main`-Methode folgende Aktionen durch:

- Erzeugen Sie ein neues Portal, lesen Sie die Datenbankdatei `Hotels.mupdb` ein und geben Sie die Anzahl der eingelesenen IDs aus.
- Lesen Sie die Bewertungsdatei `Bewertungen.txt` ein und geben Sie die Anzahl der eingelesenen Bewertungen aus.
- Schreiben Sie das Hotel mit der ID 135693 in die Datei `Hotel135693.txt`.