Inhalt der Vorlesung

1. Klassen und Objekte
2. Vererbung
3. Enums, Wrapper und Autoboxing
4. Interfaces
5. Generics
6. Exceptions
7. Polymorphismus
8. Grafische Oberflächen
9. Streams und Lambda Expressions
10. Leichtgewichtige Prozesse – Threads

Kapitel 6: Exceptions

Inhalt

Kapitel 6: Exceptions

Lernziele

[LZ 6.1] Das Fehlerbehandlungskonzept von Java erläutern können
[LZ 6.2] Eine lokale Exception-Behandlung defineren können
[LZ 6.3] Eine Exception weiterreichen können
[LZ 6.4] Benutzer-definierte Exception-Klassen entwickeln können
[LZ 6.5] Exceptions in Fehlersituationen selbst auslösen können
[LZ 6.6] Den finally-Block zielgerichtet einsetzen können

6. Exceptions

6.1 Fehlerbehandlung in Java

Anders als „C“ besitzt Java ein integriertes Konzept für die Behandlung von Laufzeitfehlern: es gibt den Standard-Ablauf einer Methode, in dem die fachliche Datenverarbeitung geschieht. Getrennt davon wird jede Art von Fehlerbehandlung, die während des Standard-Ablaufs notwendig wird, als Ausnahme-Behandlung (*exception handling*) entwickelt. Ein Laufzeitfehler wird daher *Exception* (Ausnahme) genannt.

Beispiele für Exceptions sind:

- NullpointerException bei Methodenaufruf auf null-Referenz:

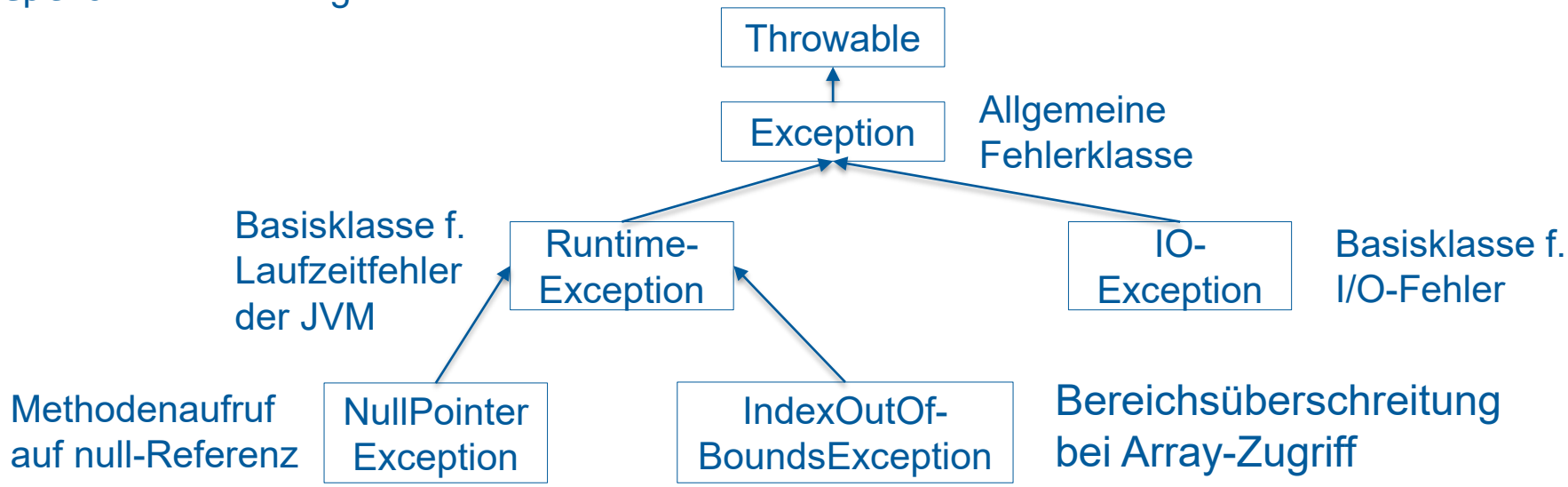
```
Student s;
s.setMatrikelnummer(4711); // Fehler: s ist null!
```

- Bereichsüberschreitung bei Array-Zugriff: ArrayIndexOutOfBoundsException

```
int[] intArray = { 1, 2, 3 };
intArray[3] = 4; // Index 3 ungültig!
```

Wird eine Exception „geworfen“, so wird die Ausführung der aktuellen Anweisung abgebrochen. Für das Weitere gibt es nun zwei Möglichkeiten: Exceptions können in der Methode, in der sie auftreten, abgefangen und behandelt oder an die aufrufende Methode weitergeworfen werden. Das Abfangen geschieht mit Hilfe einer try-catch-Konstruktion, siehe Abschnitt 6.2.2.

Zur Beschreibung von Laufzeitfehlern gibt es in Java spezielle Klassen, die alle von `Throwable` erben. `Throwable` („werfbar“) als Basisklasse mit wichtigen Informationen (z.B. Stacktrace) für alle abgeleiteten Klassen wird im Gegensatz zu `Exception` nicht direkt genutzt:



Der Name der `Exception-Klasse` deutet in der Regel auf die Art des Fehlers hin. Eigene, anwendungsspezifische Exceptions werden typischerweise als Unterklasse von `Exception` definiert. Fehler beim Zugriff auf eine Datenbank könnten beispielsweise in Form einer selbstdefinierten `DBException` geworfen werden, die dann DB-spezische Fehlercodes enthält.

Jede Methode muss angeben, ob und welche Exceptions sie möglicherweise wirft (engl. *throws*). Als Beispiel sei der Konstruktor der Klasse java.io.FileInputStream, die zum Lesen von Dateien verwendet wird, genannt:

```
public FileInputStream(File file) throws FileNotFoundException {
   // Die Implementierung wirft eine FileNotFoundException, wenn
   // der Parameter file keine existierende Datei beschreibt
}
```

FileInputStream ist eine vordefinierte Klasse aus der Java Laufzeitbibliothek. Für selbstentwickelte Methoden gilt Obiges jedoch analog.

(Un)Checked Exceptions
Der Java-Compiler prüft für jede Methode, ob diese Methoden aufruft, die Exceptions werfen können. Ist dies der Fall, so wird geprüft, ob die Exception weitergereicht wird (vgl. 6.2.1) oder die Exception in Form eines try-catch-Konstrukts selbst behandelt wird (vgl. 6.2.2). Eine Ausnahme stellten RuntimeExceptions und deren Ableitungen dar: diese werden typischerweise von der JVM geworfen und werden entsprechend nicht überprüft.
Bei Exceptions und deren Ableitungen spricht man von *checked exceptions*, bei *RuntimeExceptions* und deren Ableitungen entsprechend von *unchecked exceptions.*

6. Exceptions

6.2.1 Exceptions weiterreichen

In allen Fällen, wo man eine Exception nicht sinnvoll behandeln kann, ist es besser, diese weiterzuwerfen. Dazu genügt es, im Methodenkopf eine throws-Deklaration vorzunehmen. Nachfolgend ein Beispiel zum Einlesen einer Textdatei: Auftretende Exceptions werden zweimal (aus liesDateiEin bzw. main) weitergeworfen und führen zu einem Stacktrace in der Konsole.

```
public class TextdateiLeser {
    public String liesDateiEin(String dateiname) throws FileNotFoundException, IOException {
        String ergebnis = "";
        BufferedReader input = new BufferedReader(new FileReader(
            new File(dateiname)));
        String line;
        while ((line = input.readLine()) != null)
            ergebnis += line + "\n";
        return ergebnis;
    }
    static public void main(String args[]) throws Exception {
        String dateiText = new TextdateiLeser().liesDateiEin("C:\\tmp\\test.txt");
        System.out.println(dateiText);
    }
}
```

Kann eine im Rahmen einer Methodenausführung auftretende Exception lokal sinnvoll behandelt werden, so kann dies mit Hilfe eines try-catch-Konstrukts geschehen. Beispiel einer try-catch-Konstruktion mit mehreren catch-Klauseln:

```
try {
    // Ablauf, der Exceptions werfen kann
} catch (NumberFormatException nfe) {
    // Fehlerbehandlung 1
} catch (IOExcption ioe) {
    // Fehlerbehandlung 2
} catch (NullpointerException | ArithmeticException  ex) {
    // ab Java 7: multi-catch – Exceptiontypen durch | getrennt
} catch (Exception e) {
    // allgemeinste Exception ➔ sollte letzter catch-Block sein!
}
```

Was kann man in einem catch-Block tun (e sei die Exception-Referenz) ?

- Ausnahme ignorieren (leere Anweisung)
- Ausnahme auf die Konsole ausgeben, z.B. mit `e.printStacktrace()`
- Ausnahme mit `throw` weiterwerfen: `throw e;`
- Neue Ausnahme erzeugen und werfen: `throw new Exception(e.getMessage());`
- Ausnahme „vernünftig“ behandeln, d.h. Fehlersituation auflösen

Beispiel: Division durch eingelesene Zahl, die Kommentare enthalten wichtige Erläuterungen

```
public class Teiler {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in); // Scanner* für Standardeingabe
        int zahl = 4711;

        while (true) { // Endlos: ganze Zahl einlesen und 4711/<zahl> ausgeben
            try {
                System.out.print("Teiler: ");
                int i = sc.nextInt(); // int einlesen und durch i teilen
                System.out.println(zahl + " / " + i + " = " +  (zahl / i));
             } catch (ArithmeticException e) {
                System.out.println("Fehler: Division durch 0!");
             } catch (InputMismatchException ime) {
                // Scanner-Fehler: Eingabetext nicht in eine Zahl umwandelbar!
                System.out.println("Fehler: keine Zahl eingegeben!");
                String s = sc.next(); // Eingabepuffer leeren
             }
}}}
```

*Scanner: wandelt einen Eingabestrom in sinnvolle Einheiten (hier: int mit nextInt()) um.

Java bietet eine Menge vordefinierter Exceptions für Standard-Fehler an, die während der Programmausführung typischerweise auftreten, wie z.B. die NullPointerException. Grundlegende Informationen wie ein *Stacktrace* und ein Attribut *message* (Fehlermeldung) sind in der Basisklasse enthalten, wenn man darüberhinaus Informationen bzw. Methoden anbieten möchte, um z.B. detailliertere Informationen zum Fehler mitteilen zu können, definiert man eigene Exeption-Klassen durch Ableitung von Exception.

```
class MeineException extends Exception {
    private String detailMessage;

    public MeineException(String message, String detailMessage) {
        super(message); // message kommt von Oberklasse und wird hier initialisiert
        this.detailMessage = detailMessage;
    }
    public String getDetailMessage() { return detailMessage; }
}
```

Entwurfsideen:
(1) Fehlerbeschreibung durch message und detailMessage
(2) message wird von Exception geerbt!

Obige Klasse ergänzt Exception um ein Attribut detailMessage und einen Getter dafür. Im Konstruktor muss der Oberklassenkonstruktor mit der Fehlermeldung aufgerufen werden, bevor die detailMessage gespeichert wird. Im nächsten Abschnitt wird erläutert, wie Exceptions (vorgegebene und selbstentwickelte) geworfen werden können.

Beispiel einer kleinen Exception-Hierarchie: DBException erbt von ApplicationException

```
public class ApplicationException extends Exception {
    private String userMessage; // wird dem Benutzer angezeigt
    private String internalMessage; // interne technische Fehlermeldung

    public ApplicationException(String userMessage,
            String internalMessage) {
        this.userMessage = userMessage;
        this.internalMessage = internalMessage;
}}
```

ApplicationException speichert
- userMessage
- internalMessage
- message (geerbt von Exception)

```
public class DBException extends ApplicationException {
     private int dbErrorCode;

    public TechnicalException(int errorCode,
            String userMessage, String dbMessage) {
        super(userMessage, dbMessage);
        this.dbErrorCode = errorCode;
    }
}
```

DBException speichert
- dbErrorCode
- alle Attribute von ApplicationException

Exceptions bieten u.a. folgende Methoden an:
- getMessage: liefert die eigentliche Fehlermeldung
- printStackTrace: Ausgabe eines Stacktrace auf die Konsole

Über den Konstruktor Exception(String message) wird ein Exception-Objekt erzeugt und die Fehlermeldung wird initialisiert.

Exceptions werden ausgelöst, indem ein Exception-Objekt erzeugt wird (wenn noch keines vorhanden ist) und anschließend mit der throw-Anweisung „geworfen“ wird:

```
public String parseInput() throws Exception {
    String input = Console.getInputAsString();

    if (input == null || input.length() == 0) {
        throw new Exception("Leereingabe nicht erlaubt!");
    }
    return input;
}
```

Nach Ausführung einer throw-Anweisung wird die Ausführung der Methode unterbrochen. Gibt es einen umschließenden passenden catch-Bock, so wird dieser ausgeführt, andernfalls wird die Exception an die aufrufende Methode weitergereicht (s. oben).

Ein try-catch-Konstrukt kann um eine finally-Klausel ergänzt werden, wenn vor dem Verlassen einer Methode (ob „normal“ oder mit einer Exception) gewisse „Aufräumarbeiten wie z.B. das Schließen einer Datenbank-Verbindung vorgenommen werden sollen.

Beispiel:

```
try {
    datenbankVerbindungOeffnen();
    heikleDbOperation();
} catch (SQLException e) {
    ...
    // Fehler loggen -- evtl. beheben
} finally {
    datenBankVerbindungSchliessen();
}
```

Beachte: der finally-Block wird immer ausgeführt!