Westfälische
Wilhelms-Universität
Münster

Programmieren in Java

Fehlerbehandlung und Ein- und Ausgabe

wissen.leben
WWU Münster

Patrick Förster, Thorsten Hovenga

ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

Fehler = Exceptions?

- Die Exception- bzw. Ausnahmebehandlung in Java ist eine spezielle Form der Fehlerbehandlung
- Typischerweise treten beim Programmieren folgende Fehler auf:
- **Syntaktische Fehler:** Diese werden bereits vom Compiler erkannt und das geschriebene Programm kann erst nach der Behebung vom Compiler übersetzt werden
- **Logische Fehler und Bugs:** Sind meistens schwer zu lokalisieren (z.B. + und - vertauscht). Können durch systematisches Ausgeben eigener Debug-Informationen oder mit Hilfe eines Debuggers behoben werden.
- **Laufzeitfehler:** Meistens liegt die Fehlerquelle außerhalb des Programms. Hier können Benutzerangaben oder Ressourcenanforderungen(z.B. Dateien) die Ursache sein. Diese Fehler müssen im Programm abgefangen bzw. behandelt werden.

wissen.leben WWU Münster

Fehler an Ort und Stelle vermeiden

- Grundsätzlich sollten Fehler an dem Ort Ihres Auftretens vermieden oder behandelt werden
- Beispiel: Bevor man auf die Argumente eines Programmaufrufs zugreift sollte man sich vergewissern das diese auch vorhanden sind

```
public static void main(String[] args) {
if (args.length != 1){
System.out.println("\nFalsche Anzahl Argumente!");
System.out.println("Aufruf: java Programmname <Argument>");
System.exit(0);
}
else
System.out.println(args[0]);
}
```

wissen.leben WWU Münster

Fehlerbehandlung durch Codierung in Rückgabewerten

- Wenn eine Methode einen Fehler „bemerkt“ kann sie durch spezielle Rückgabewerte außerhalb des normalen Wertebereichs eine Fehlerbehandlung an dem Ort ermöglichen an dem sie aufgerufen wurde.
- Das führt zu Problemen wenn der Rückgabewert kein primitiver Datentyp ist sondern zum Beispiel ein Objekt einer selbst definierten Klasse
- Dieses Vorgehen erschwert die Lokalisierung von Fehlern da scheinbar ein vernünftiges Ergebnis zurückgegeben wird

```
public static double division(double zaehler, double nenner){
        if (nenner == 0.0)
                return Double.NaN;
        else
                return zaehler/nenner;
        }
```

```
quotient = division(zaehler,nenner);
if (Double.isNaN(quotient)){
System.out.println("Fehler: Division durch Null");
}
else{
System.out.println("Divisionsergebnis: "+quotient);
}
```

wissen.leben WWU Münster

Exceptions

- Die Behandlung von Fehlern mit Exceptions soll die vorher genannten Vorgehensweisen nicht ersetzen.
- Exceptions dienen dazu das Auftreten und erkennen eines Fehlers räumlich im Code zu trennen
- Entweder soll der Code für eine bessere Übersicht nicht durch aufwendige Fehlerbehandlung unterbrochen werden
- Oder eine sinnvolle Fehlerbehandlung ist am ort des Auftretens nicht möglich weswegen der Fehler an einen anderen Programm-Block oder eine aufrufende Methode weitergegeben wird

- Beispiel:

```
// Wandelt einen String in ein double-Wert um

double zahl = Double.parseDouble(“123.45“);

double zahl = Double.parseDouble(“23 Euro“); // Fehler

// Fehlerbehandlung nicht innerhalb von parseDouble

// Deshalb Fehlerbehandlung angepasst auf eigen Bedürfnisse direkt in
der aufrufenden Methode
```

wissen.leben WWU Münster

Exceptions erzeugen

- Damit eine Fehlerbehandlung nötig wird muss ein Fehler auftreten oder eine Fehlerquelle bereits vor Ausführung des Programms bekannt sein
- Fehler wird abgefangen durch vorher erläuterte Methoden und mit dem Schlüsselwort `throw` wird eine Exception ausgelöst
- Eine Exception ist ein Objekt der Klasse `Exception` oder einer abgeleiteten Klasse von `Exception`
- Dieses Verhalten wird in der Signatur der Methode bereits angekündigt mit `throws <Exceptionliste>`
- Der Ablauf des Programms wird unterbrochen und an der Stelle der Fehlerbehandlung fortgesetzt.

```
public static double division(double zaehler, double nenner) throws
Exception{

if (nenner == 0.0)

throw new Exception();

else

return zaehler / nenner;

}
```

wissen.leben WWU Münster

Exceptions abfangen

- Die ausgelöste Exception wird von „innen nach außen“ zu den umliegenden Blöcken und Methoden weitergeleitet
- Wird auf dem Weg ein `try`-Block erreicht für den ein zur Exception passender `catch`-Handler definiert ist, wird die Exception von diesem abgefangen

```
public static void main(String[] args) {
double zaehler=0,nenner=0,bruch=0;

try{
     zaehler = 23.56;
     nenner = 3.0;
     bruch = division(zaehler,nenner);
     System.out.println("\n\t"+zaehler+" : "+nenner+" = "+bruch);
     //Aufruf der Exception auslöst
     zaehler = 23.56;
     nenner = 0.0;
     bruch = division(zaehler,nenner);
     System.out.println("\n\t"+zaehler+" : "+nenner+" = "+bruch);
     }
     catch (Exception e){

     System.out.println("\n\t Fehler: Exception wurde abgefangen");
          }
}
```

wissen.leben WWU Münster

Exception-Hierarchie

- Es können mehrere catch-Handler definiert werden wenn verschiedene Exceptions vorkommen können
- Exception-Objekte können implizit zu ihrem Basisobjekt umgewandelt werden, sodass der Aufruf von `catch (Exception e)` alle Exceptions abfängt. (Ist das ein Vor- oder Nachteil?)
- Catch-Handler werden in der Reihenfolge ihrer Definition durchsucht. Deshalb mit der Definition des speziellsten Handlers anfangen damit keine Verdeckung durch einen Allgemeineren erfolgt.
- Methoden von Exceptions stammen von der Basisklasse von `Exception`; `Throwable`
- Methoden können im Catch-Handler genutzt werden durch Zugriff auf Objektreferenz `e`
- Vorhandene Methoden sind:

```
Throwable(String meldung) //Konstruktor erlaubt die Übergabe einer Meldung

Throwable getCause() // zum Verketten von Exceptions (nächste Folie)

String getMessage() // Meldungstext einer Exception

Void printStackTrace() // Beschreibung plus Liste der Methoden auf dem Stack

String toString() // Beschreibung der Exception (Klassennamen,Meldungstext)
```

wissen.leben WWU Münster

Weiterleitung von Exceptions

- Exceptions könnn auch weitergeleitet werden ohne dass sie behandelt werden
- Try- und catch-Block werden in dem Fall weggelassen und in der aufrufenden Methodensignatur wird mit throws ‹Exceptiontyp› weiter „nach oben“ angezeigt, dass die Exception behandelt werden soll
- Exceptions vom Typ RuntimeException müssen nicht zwingend behandelt oder angezeigt werden (kann nützlich sein um Konflikte bei geerbten Methoden zu vermeiden)

```
public static void eineMethode(){
        try{
                throw new Exception("Originale Exception!");
        }
        catch (Exception e){
                throw new RuntimeException(e);
        }
}
public static void main(String[] args) {
        try {
                eineMethode();
        }
        catch (Exception e){
                System.out.println(e.getCause().getMessage());
        }
}
```

wissen.leben WWU Münster

Eigene Exceptions

- Wenn es keine passende Exception für einen Fehler gibt sollte eine Eigene definiert werden
- Das bietet die Möglichkeit der Fehlermeldung weitere Informationen hinzuzufügen

```
public class NullDivisionException extends Exception {
private double zaehler;

public double getZaehler(){
return zaehler;
}
public NullDivisionException(){
}
public NullDivisionException(String meldung){
super(meldung);
}
public NullDivisionException(String meldung, double zaehler){
super(meldung);
this.zaehler = zaehler;
}
}
```

wissen.leben WWU Münster

Aufruf eigene Exception

```
  public class EigeneExceptions {

  public static double division(double zaehler, double nenner) throws
NullDivisionException{
  if (nenner == 0.0)
          throw new NullDivisionException("Nulldivision",zaehler);
  else
          return zaehler / nenner;
  }
  …

  catch (NullDivisionException e){
          System.out.println(e.getMessage());
          System.out.println("Zaehler war gleich: "+e.getZaehler());
          }
  }
  }
```

Anmerkung: Ausser try- und catch-Block gibt es noch einen finally-Block in dem Anweisungen abgelegt die auch ausgeführt werden wenn eine Exception ausgelöst wurde (z.B. freigeben von Ressourcen)

wissen.leben WWU Münster

Dateien behandeln mit File

- Um Dateien zur Ein- und Ausgabe nutzen zu können wird zunächst ein Verfahren benötigt um mit dem Dateisystem umzugehen und Informationen über Dateien und Verzeichnisse zu bekommen
- Dazu gibt es in Java die Klasse `java.io.File`. `File` dient nicht zum eigentlichen Lesen und Schreiben von Dateien
- `File datei = new File(„vorlesung_3.txt“);`
- Die wichtigsten Methoden von `File` sind:
- `String getName()` – Datei/Verzeichnisname wie beim Anlegen des Objekts
- `String getPath()` – Pfad wie beim Anlegen des Objekts
- `String getAbsolutePath()` – absoluter Pfad, wenn nicht angegeben wird das Arbeitsverzeichnis genutzt
- `String getCanonicalPath()` – absoluter Pfad + Bereinigung
- `String getParent()` – Verzeichnis in dem das Objekt ist, null falls unbekannt

wissen.leben WWU Münster

Dateien erzeugen/löschen

- File-Objekte sind zunächst keine Dateien. Wenn man entsprechende File-Objekte definiert hat kann man allerdings auf echte Dateien zugreifen:
- `boolean canRead(), boolean canWrite()` – Überprüft Lese/Schreibrechte
- `boolean createNewFile()` – `true` wenn die Datei noch nicht existierte und neu angelegt wurde
- `boolean delete()` – `true` wenn die Datei gelöscht werden konnte
- `boolean renameTo(File neu)` – `true` wenn die Datei in neu umbenannt werden konnte
- `long length()` – gibt die Länge der Datei in Bytes zurück

wissen.leben WWU Münster

Das Stream-Konzept

- Die Vorstellung von Streams ist ein Fluss eines Datenstroms(stream) von bytes zwischen einem Sender und einem Empfänger. Je nachdem wer Sender und Empfänger ist unterscheidet man in `java.io.InputStream` und `java.io.OutputStream.(Bsp. FileInputStream)`
- Basistyp dieser Klasse ist `byte`. Da Dateien häufig auf Text basieren gibt es zu den Stream-Klassen entsprechende Zeichenbasierte Klassen mit `Reader/Writer` statt `Stream`. (`Bsp. FileReader, FileWriter`)
- Methoden von InputStream/OutputStream:
- `int read/write()` – liest/schreibt das nächste byte; Rückgabe als int oder -1 bei Misserfolg
- `int read/write(byte[] b)` – Liest/schreibt eine Anzahl bytes in das Array b/aus dem Array b; Rückgabe Anzahl an Bytes oder -1
- `void close()` – Schließt den Stream und gibt Ressourcen frei

wissen.leben
WWU Münster

Properties-Dateien

- Properties-Dateien verendet man häufig um Konstanten abzulegen damit bei einer Änderung nicht immer neu compiliert werden muss
- Klasse `java.util.Properties`
- Dateien mit FileInputStream/FileOutputstream öffnen
- Zugriff mit `setProperty(),getProperty(String prop), load( Inputstream Eingabe), store(OutputStream ausgabe, String Kommentar), Enumeration propertyNames()`

wissen.leben
WWU Münster