Java – Fehlerbehandlung

Fehlerbehandlung

In jedem nicht-trivialen Programm kann es während der Laufzeit zu Fehlersituationen kommen.
Dabei handelt es sich nicht unbedingt um Programmierfehler:

z. B.: Programm soll Daten in einer Datei abspeichern;
Benutzer gibt falschen Dateinamen an;
Datei kann wegen fehlender Zugriffsrechte nicht angelegt werden.

Fehlerbehandlung

Problem:

Größere Programme sind modular aufgebaut;
Fehlersituationen treten „innen“ auf:

main ruft methode1 auf,
methode1 ruft methode2 auf,
methode2 ruft methode3 auf,
methode3 ruft methode4 auf,
in methode4 wird festgestellt, daß die Datei nicht angelegt werden kann.

Fehlerbehandlung

(Zu) einfache Lösungen:

Abbruch des Programms:

für interaktive Programme inakzeptabel.

Fehlermeldung und lokale Fehlerbehandlung:

Fehlermeldung: wohin?
(Terminal, Statuszeile, neues Fenster?)
Fehlerbehandlung: meist nicht lokal möglich.

Fehlerbehandlung

Umständliche Lösung:

Error Code (z. B. als Return-Wert):

Error Code muß typkorrekt, aber von normalen Ergebnissen unterscheidbar sein.
Weiterverarbeitung ist umständlich.
Programmierer, die eine Methode benutzen, ignorieren Error Codes häufig ($\rightsquigarrow$ fehlerhafte Ergebnisse, Programmabsturz).

Exceptions

Elegantere Lösung (z. B. in Ada, C++, Java):

Exception:

Ereignis, das während der Laufzeit eines Programms eintritt und den normalen Kontrollfluß unterbricht.

z. B.: Hauptspeicherüberlauf,
Division durch 0,
Zugriff auf nicht-existierendes Arrayelement,
Datei kann nicht geöffnet werden.

Exceptions

Idee:

Momentan laufende Methode erzeugt ein Exception-Objekt (enthält u.a. Information über die Art des Ereignisses).

Laufzeitsystem sucht auf dem Aufrufstack (rückwärts von der momentanen Methode aus) den ersten passenden Exception-Handler (= Code, der das Exception-Objekt übernimmt und damit weiterarbeitet).

Falls kein passender Exception-Handler gefunden wird, wird das Java-Programm beendet.

Exceptions fangen

Exceptions bearbeiten („fangen“):

```
try {
    statements1
} catch (...) {
    statements2
}
```

`statements1` wird ausgeführt.

Gibt es während der Ausführung (direkt in `statements1` oder indirekt innerhalb einer der aufgerufenen Methoden) eine Exception, dann geht es unmittelbar mit der Ausführung von `statements2` weiter.
(Anderenfalls wird `statements2` nicht ausgeführt.)

Exceptions fangen

Erweiterung des Konzepts:

Während der Ausführung von `statements1` können verschiedene Arten von Fehlern oder Ausnahmezuständen auftreten.
Auf diese soll verschieden reagiert werden, z. B.:

- Datei kann nicht angelegt werden:
  ⇝ Benutzer nach neuem Dateinamen fragen.
- Fehlerhafter Arrayzugriff:
  ⇝ Programmierfehler: Daten falls möglich retten, Fehlerprotokoll ausgeben, Programm abbrechen.
- Hauptspeicherüberlauf:
  ⇝ wahrscheinlich keine sinnvolle Gegenmaßnahme möglich: Programm abbrechen.

Exceptions fangen

Erweiterung des Konzepts:

```
try {
    statements1
} catch (IOException e1) {
    statements2a
} catch (IndexOutOfBoundsException e2) {
    statements2b
}
```


Exceptions fangen

Erweiterung des Konzepts:

Falls das Exception-Objekt zur Klasse `IOException` (oder einer Unterklasse) gehört, wird `statements2a` ausgeführt. Innerhalb von `statements2a` kann auf das Exception-Objekt unter dem Namen `e1` zugegriffen werden.

Falls das Exception-Objekt zur Klasse `IndexOutOfBoundsException` (oder einer Unterklasse) gehört, wird `statements2b` ausgeführt. Innerhalb von `statements2b` kann auf das Exception-Objekt unter dem Namen `e2` zugegriffen werden.

Falls das Exception-Objekt zu keiner der beiden Klassen gehört (z.B. `OutOfMemoryError`), dann wird auf dem Aufrufstack rückwärts weiter ein passender Exception-Handler gesucht.

Exceptions fangen

Zweite Erweiterung des Konzepts:

```
try {
    statements1
} catch (IOException e1) {
    statements2a
} catch (IndexOutOfBoundsException e2) {
    statements2b
} finally {
    statements3
}
```

Exceptions fangen

Zweite Erweiterung des Konzepts:

`statements3` wird auf jeden Fall ausgeführt,
egal ob `statements1` regulär beendet wird,
oder eine `IOException` oder `IndexOutOfBoundsException`
auftritt (die gefangen wird),
oder eine andere Exception auftritt (die nicht gefangen wird).

Übliche Anwendung: Aufräumarbeiten.
z. B.: Dateien in konsistenten Zustand bringen,
Bildschirmfenster aufräumen,
Netzwerkverbindung schließen.

Exceptions werfen

Exceptions erzeugen („werfen“):

entweder implizit,
z. B.: Division durch 0,
Zugriff auf nicht-vorhandenes Arrayelement `a[-1]`,
Senden einer Nachricht an `null`,

oder explizit:

```
throw new IOException();
throw new IOException("Falsches Datenformat");
```


Exceptions werfen

```
static public void main(String[] argumente) {
    System.out.println("main<1>");
    try {
        System.out.println("main<2>");
        System.out.println(letztes(argumente));
        System.out.println("main<3>");
    } catch (IndexOutOfBoundsException e) {
        System.out.println("main<4>");
    } catch (NullPointerException e) {
        System.out.println("main<5>");
    } finally {
        System.out.println("main<6>");
    }
    System.out.println("main<7>");
}
```


Exceptions werfen

```
static public int letztes(String[] array) {
    String s;
    int n;
    System.out.println("letztes<1>");
    s = array[array.length-1];
      // wirft IndexOutOfBoundsException,
      // falls array Länge 0 hat.
    System.out.println("letztes<2>");
    n = Integer.parseInt(s);
      // wirft NumberFormatException,
      // falls s keine Zahl darstellt.
    System.out.println("letztes<3>");
    return n;
}
```

Catch or Specify

Die Exceptions, die (direkt oder indirekt) innerhalb einer Methode geworfen, aber nicht gefangen werden, sind ein Teil der Schnittstelle dieser Methode.

Sie müssen darum in Java deklariert werden:

```
public String f(int n)
        throws TooBigException, TooSmallException {
    ...
}
```


Catch or Specify

Eigentlich müßte dann jede Methode, in der auf ein Array zugegriffen wird, mit `throws IndexOutOfBoundsException` deklariert werden.
(Das wäre lästig.)

Außerdem müßte jede Methode, in der ein neues Objekt angelegt wird, mit `throws OutOfMemoryError` deklariert werden.
(Das wäre noch lästiger.)

Darum gibt es zwei Ausnahmen von dieser Regel.

Klassenhierarchie: Throwable

Klassenhierarchie:

```
Throwable
   Error
      OutOfMemoryError
      StackOverflowError
      ...
   Exception
      RuntimeException
         IndexOutOfBoundsException
         NullPointerException
         ...
      FileNotFoundException
      IOException
      ...
```

`Error`:
    schwerwiegende Fehler des Java-Laufzeitsystems. (Brauchen nicht deklariert zu werden.)

`RuntimeException`:
    Exceptions, die von primitiven Operationen geworfen werden können, und eigentlich nicht auftreten sollten, aber fast überall auftreten können. (Brauchen nicht deklariert zu werden.)

Java – Eingabe und Ausgabe

Ein-/Ausgabe

Stream:

Objekt, von dem sequentiell gelesen oder auf das sequentiell geschrieben werden kann.

Kann z. B. mit einer Datei verbunden sein (oder auch: Netzverbindung, String, anderer Stream).

Ein-/Ausgabe

Historisches Problem:

In Java 1.0:

Streams zur `byte`-weisen Ein-/Ausgabe: `XyzInputStream`, `XyzOutputStream`.

In Java 1.1 *zusätzlich*:

Streams zur `char`-weisen Ein-/Ausgabe: `XyzReader`, `XyzWriter`.

Faustregel: Wenn möglich, `Reader` und `Writer` benutzen. `InputStream` und `OutputStream` sind aber manchmal immer noch notwendig; beispielsweise gehört `System.out` (Standardausgabe) zur Java-1.0-Klasse `PrintStream`.

Ein-/Ausgabe

Java-1.0-Klassen zur byteweise Ein-/Ausgabe:

`InputStream` (z.B.: `System.in`)

- `FileInputStream`
- `BufferedInputStream`
- ...

`OutputStream`

- `FileOutputStream`
- `BufferedOutputStream`
- `PrintStream` (z.B.: `System.out`, `System.err`)
- ...

Ein-/Ausgabe

Java 1.1: Klassen zur zeichenweise Ein-/Ausgabe.

`Reader`

- `InputStreamReader`
  - `FileReader`
- `BufferedReader`
- `StringReader`
- ...

`Writer`

- `OutputStreamWriter`
  - `FileWriter`
- `BufferedWriter`
- `StringWriter`
- `PrintWriter`
- ...

Ein-/Ausgabe

Textdatei lesen: zeichenweise, ungepuffert.

```
int ch;
FileReader in;

try {
    in = new FileReader("file.txt");
    while( (ch = in.read()) != -1 ) {
        ...
    }
    in.close();
} catch (IOException e) {
    System.out.println("Read error: " + e.getMessage());
}
```


Ein-/Ausgabe

Textdatei lesen: zeilenweise, gepuffert.

```
String line;
BufferedReader in;

try {
    in = new BufferedReader(new FileReader("file.txt"));
    while( (line = in.readLine()) != null ) {
        ...
    }
    in.close();
} catch (IOException e) {
    System.out.println("Read error: " + e.getMessage());
}
```