Informatik I

Prof. Dr. Christian Pape

Exceptions

Inhalt

- Fehlerbehandlung ohne Exceptions
- Exceptions
- Dateien

Fehlerbehandlung ohne Exceptions

- Problem
  - Methode wird aufgerufen
  - In der Methode tritt ein Fehler auf
  - In der Methode kann der Fehler nicht sinnvoll behandelt werden
  - Wie kann das die Methode aufrufende Programm diesen Fehler bemerken und darauf reagieren?
- Beispiel: Dateien

Fehlerbehandlung ohne Exceptions

- Programmiersprache C
  - Fehler werden durch Rückgabewerte angezeigt („funny numbers“ [Java Spec.])

```
FILE *fp  = fopen("liebesgeschichten.txt", "r");
// file ist ein Referenztyp
// die Speicheradresse der Referenz kann direkt
// abgefragt werden (implizite Konvertierung zu int)
char zeile[100]; // Zeichenkette mit 100 Zeichen
if (file != null) { // null: Fehler beim Öffnen
  // lese Datei Zeile für Zeile ein
  // fgets gibt im Fehlerfall (und Ende Datei) null zurück
  while ( (zeile = fgets(zeile, 100, fp)) != null) {
    printf(“%c“, zeile);
  }
} else {
  printf(“Konnte die Liebesgeschichten nicht lesen.“);
}
fclose(fp);
```

Fehlerbehandlung ohne Exceptions

- Beispiel: Rechnerübung Datum
  - Bei fehlerhaftem Datum wird immer -1 bei getTag(), … zurückgegeben
  - Methode isKorrekt(), die anzeigt, dass Datum fehlerhaft ist

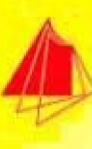
```
Datum datum = new Datum(1,65, 2000);

if (datum.isKorrekt()) {
   // mache etwas mit dem datum
} else {
   // Bestrafe Benutzer für die Angabe
   // eines falsches Datum
}
```

Fehlerbehandlung ohne Exceptions

- Probleme
  - Funktionen, die -1 (null) oder jeden anderen Wert als gültiges Ergebnis zurückgeben, können so nicht behandelt werden
  - Objektmethoden, die Fehlercodes zurückgeben können bei statischen Methoden nicht verwendet werden
  - Rückgabewerte für Fehler werden von Programmierern oft ignoriert (oft nicht ersichtlich, ob und welcher Wert einen Fehlerzustand anzeigt)
  - Es kann nur ein Code (Zahl), aber nicht zusätzliche Information (Fehlertext) zurückgegeben werden
  - Programme werden unübersichtlich (keine gute Trennung zwischen Fehlerbehandlung und Normalablauf)
- Lösung (C++, Java, C#, …)
  - Methoden haben **zweiten** Rückgabewert : Instanz von Exception
  - Auch Konstruktor kann eine Exception zurückgeben
  - Exceptions sind Objekte
  - Syntax für „Rückgabe“ statt mit **return** durch **throw**
  - Zugriff auf Exception eines Aufrufs mit speziellen Kontrollanweisungen try – catch - finally

Inhalt

- Fehlerbehandlung ohne Exceptions
- Exceptions
- Dateien



Exceptions

- Im Fehlerfall
  - Konstruktur einer Exception aufrufen
    new NullPointerException(“Mächtig großer Fehler“);
  - Exception mit **throw** nach „außen werfen“

```
public Person(String name, Adresse adresse) {
  // stelle sicher, dass name und adresse vorhanden sind
  if (name == null) {
    throw new NullPointerException(“Kein Name angegeben“);
  }
  if (adresse == null) {
    throw new NullPointerException(“Keine Adresse angegeben“);
  }
  this.name = name;
  this.adresse = adresse;
}
```

Konstruktor wird hier abgebrochen
Programmkontrolle geht zur aufrufenden Stelle über

Exceptions

- Es existieren verschiedene Exceptions

```
public static long fakultaetBerechnen(long n) {
  long fakultaet = 1;

  if (n < 0) {
    throw new IllegalArgumentException(“Negativer Wert verboten“);
  }

  while (n > 0) {
    fakultaet = fakultaet * n;
    n--;
  }

  return fakultaet;
}
```

Exceptions

- Checked exceptions (gibt es in C++, C# nicht)
  - Alle Exceptions mit Oberklasse Exception
  - **Müssen** vom Programmierer behandelt werden (Compiler prüft dies)
- Unchecked exceptions (analog in C++, C#)
  - Alle Objekte mit Oberklasse Throwable aber nicht Exception
  - **Können** vom Programmierer behandelt werden

Inhalt

- Fehlerbehandlung ohne Exceptions
- Exceptions
  - unchecked exceptions
  - checked exceptions
- Dateien

Exceptions

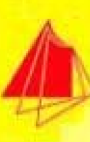
- Zurückgegeben Exception kann nicht mit zuweisen „entgegengenommen" werden
- try-catch-Kontrollanweisung nötig: Exception wird „aufgefangen"
- Wird im try-Teil eine Exception geworfen, dann wird zum entsprechenden catch-Teil verzweigt
- Danach geht Programmkontrolle zu den Anweisungen nach dem gesamten try-catch über
- Unchecked exceptions: try-catch **optional**
- Checked exceptions: try-catch **obligatorisch**

```
try {
  long f = fakultaetBerechnen(-17); // wirft RuntimeException
  System.out.println(f);
} catch (IllegalArgumentException e) {
  // e ist Verweis auf die geworfene Exception
  System.out.println(e.getMessage());
}
```

Unchecked Exceptions

- Es kann mehrere catch geben
  - Unterschiedliche Fehlerbehandlung pro Exceptiontyp
  - Je nach Typ der Exception bei unterschiedlicher Fehlerbehandlung: ein catch pro Exception
  - Genau eine Exception-Deklaration pro catch nötig
  - Erste „passende“ Typ von oben nach unten wird ausgeführt
  - Gibt es keinen passenden catch-Teil, so wird zur Anweisung hinter gesamten try-catch gesprungen

```
int n = 10;
try {
  long f = fakultaetBerechnen(n);
  System.out.println(f);
  System.out.println(new Datum(1,-12, 2000));
} catch (IllegalArgumentException e) {
  System.out.println(e.getMessage());
} catch (NullPointerException e) {
  System.out.println(e.getMessage());
}
```



Exceptions

- Exception in fakultaetBerechnen(n)
  - new Datum(1, -12, 2000) wird nicht mehr ausgeführt

```
int n = -98711;
try {
  long f = fakultaetBerechnen(n);
  System.out.println(f);
  System.out.println(new Datum(1,-12, 2000));
} catch (IllegalArgumentException e) {
  System.out.println(e.getMessage());
} catch (NullPointerException e) {
  System.out.println(e.getMessage());
}
```

Exceptions

- Bei mehreren catch geben
  - **Oberklasse vor Unterklasse angeben**
  - Sonst: Compilerfehler

```
try {
   …
} catch (RuntimeException e) {
   System.out.println(e.getMessage());
} catch (NullPointerException e) {
   System.out.println(e.getMessage());
}
```

NullPointerException ist eine RuntimeException

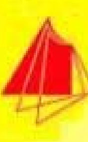
Exceptions

- Try-catch können verschachtelt werden
  - Exception in einem catch wird zum nächstem umgebende try-catch geworfen

```
try {
  long f = fakultaetBerechnen(5);
  Datum datum = null;
  try {
    datum = new Datum(1,-12, 2000);
  } catch (IllegalArgumentException e) {
    datum.setTag(15); // gibt NullPointerException
  }
} catch (NullPointerException e) {
  System.out.println(e.getMessage());
}
```

1.

2.



Exceptions

- Try-catch können verschachtelt werden
  - Nicht gefangene Exception wird zum nächstes umgebenden try-catch geworfen

```
try {
  long f = fakultaetBerechnen(5);
  Datum datum = null;
  try {
    datum = new Datum(1,-12, 2000);
  } catch (NullPointerException e) {
    //…
  }
} catch (IllegalArgumentException e) {
  System.out.println(e.getMessage());
}
```

Exceptions

- Wird eine Exception innerhalb einer Methode oder eines Konstruktors geworfen, aber nicht gefangen, so wird sie nach außen geworfen

```
public Person(String name, Adresse adresse) {
  // stelle sicher, dass name ist
  if (name == null) {
    throw new NullPointerException(“Kein Name angegeben“);
  }
´ adressse.setPerson(this); // kann NullPointerException werfen
  this.name = name;
  this.adresse = adresse;
}

 try {
   Person person = new Person(“Max“, null);
 } catch (NullPointerException e) {
   System.out.println(“Na so was“);
 }
```

Inhalt

- Fehlerbehandlung ohne Exceptions
- Exceptions
  - unchecked exceptions
  - checked exceptions
- Dateien

Exceptions

java.io

File
File(String name)
isDirectory() : boolean
listFiles() : File []

File repräsentiert eine Datei mit Name, Pfad, Grösse, …

FileReader
FileReader(File file)
FileReader(String name)

BufferedReader
FileReader(Reader reader)
readLine() : String

throws FileNotFoundException

throws IOException

Reader <<abstract>>
read() : int
read(char[] cbuf) : int

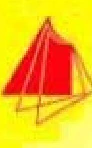
Reader zum Auslesen von Zeichen und Strings aus Dateien, Sockets, …

Analog: Writer

Exceptions

- Viele Fehler möglich
  - Datei kann nicht geöffnet werden (weil z.B. nicht vorhanden)
  - Zeichen kann nicht gelesen (z.B. keine Leseberechtigung)
  - Datei konnte nicht geschlossen werden
- IOException
  - Checked exception
  - Muss **entweder mit try – catch gefangen**
  - oder bei Methode deklariert werden

```
public void lesen(BufferedReader reader) {
  try {
    while ( reader.ready() ) {
      System.out.println( reader.readLine() );
    }
  } catch (IOException e) {
    // IOException muss gefangen werden: Compiler prüft dies
    System.out.println("Na so was ein Fehler.");
  }
}
```



Exceptions

- Falls Fehler nicht sinnvoll behandelt werden kann
  - kein try-catch
  - Exception z.B. an aufrufender Stelle fangen
  - **Bei checked exceptions: mit throws bei Methode deklarieren**
  - Unchecked exceptions können auch mit throws deklariert werden (Compiler nutzt dies aber nicht)
  - Bei mehreren nicht gefangenen checked Exceptions: alle mit Komma separiert angeben

```
public void lesen(BufferedReader reader) throws IOException {
  while ( reader.ready() ) {
    System.out.println( reader.readLine() );
  }
}
```

Exceptions

- Von einer Methode werfbaren checked Exceptions werden bei Javadoc aufgeführt (Beispiel close bei Reader)
- Bei eigenen Javadocs @throws-Tag verwenden:

  @throws IOException If an I/O error occurs

close

```
public abstract void close()
                    throws IOException
```

Closes the stream and releases any system resources associated with it. Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect.

Specified by:

`close` in interface `Closeable`

Throws:

`IOException` - If an I/O error occurs

Exceptions

- Immer externe Resource wieder freigeben
  - Datei schliessen
  - close() beim Reader aufrufen
  - Falls dies nicht passiert: Datei schreibgeschützt
  - Erst bei Ende der Virtual Maschine werden alle noch offenen Dateien, Sockets, … geschlossen

```
FileReader fileReader;
try {
  fileReader = new FileReader(“liebesgeschichten.txt“);
  ... Code für Datei lesen, IOException könnte geworfen werden
  fileReader.close(); // Datei wird nicht geschlossen
} catch (IOException e) {
  System.out.println(“Na so was“);
}
```

- Problem
  - Falls beim Lesen IOException geworfen wurde, dann wird fileReader nicht geschlossen

Exceptions

- 1. Lösungsversuch
  - Datei auch im catch schließen
- Probleme
  - Redundanz (close() ist an mehr als einem Ort)
  - Was ist, wenn gar keine IOException, sondern eine NullPointerException (oder etwas anderes) geworfen wird?
  - Datei ist dann wieder nicht geschlossen

```
FileReader fileReader;
try {
  fileReader = new FileReader(“liebesgeschichten.txt“);
  ... Code für Datei lesen, IOException könnte geworfen werden
  fileReader.close(); // Datei wird nicht geschlossen
} catch (IOException e) {
  fileReader.close(); // Datei wird jetzt auch geschlossen
  System.out.println(“Na so was“);
}
```

Exceptions

- 2. Lösungsversuch
  - Datei direkt nach try-catch schließen
- Problem
  - new FileReader() könnte IOException geworfen haben
  - dann ist fileReader == null

```
FileReader fileReader;
try {
  fileReader = new FileReader(“liebesgeschichten.txt“);
  // Datei lesen, IOException könnte geworfen werden
} catch (IOException e) {
  System.out.println(“Na so was“);
}
fileReader.close(); // fileReader könnte aber Null sein
```

Exceptions

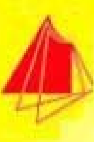
- 3. Lösungsversuch
  - Datei auch nach try-catch schließen
  - Und vorher abfragen, ob FileReader vorhanden ist
- Problem
  - Wie beim 1. Versuch, wird close() nicht aufgerufen, wenn eine Exception im try-Block geworfen wird, die nicht abgefangen wird

```
FileReader fileReader;
try {
  fileReader = new FileReader(“liebesgeschichten.txt“);
  // Datei lesen, IOException könnte geworfen werden
} catch (IOException e) {
  System.out.println(“Na so was“);
}
if (fileReader != null) {
  fileReader.close(); // fileReader könnte Null werden
}
```

Exceptions

- Lösung in Java (C#, C++)
  - Spezieller Bereich für try mit finally markiert, der **immer** ausgeführt wird
- Ausführung immer zum Schluss von try-catch
  - Falls Exception geworfen wurde: Nachdem catch ausgeführt wurde
  - Falls keine Exception geworfen wurde: Oder nachdem try vollständig ausgeführt wurde
  - finally wird auch ausgeführt, wenn
    - im catch-Block eine Exception geworfen wird
    - try / catch mit return, break oder continue „beendet“ wird

```
FileReader fileReader;
try {
  fileReader = new FileReader(“liebesgeschichten.txt“);
  ... Code für Datei lesen, IOException könnte geworfen werden
} catch (IOException e) {
  System.out.println(“Na so was“);
} finally { // wird immer zum „Schluss“ ausgeführt
  if (fileReader != null) {
    fileReader.close();
  }
}
```



Exceptions

- Catch kann nur weggelassen werden, wenn finally vorhanden ist
  - Exception in try wird dann nicht gefangen und nach aussen geworfen

```
FileReader fileReader;
try {
  fileReader = new FileReader(“liebesgeschichten.txt“);
  // Datei lesen, IOException könnte geworfen werden
} finally { // wird immer zum „Schluss“ ausgeführt
  if (fileReader != null) {
    fileReader.close();
  }
}
```

Exceptions

- close() wirft ggf. IOException
- try-catch in finally nötig
- Viel Fehlerbehandlung, aber „Normalfall“ im ersten try Block noch immer lesbar: Programm von dessen Fehlerbehandlung getrennt

```
FileReader fileReader;
try {
  fileReader = new FileReader(“liebesgeschichten.txt“);
  ... Code für Datei lesen, IOException könnte geworfen werden
} catch (IOException e) {
  System.out.println(“Na so was“);
} finally { // wird immer zum „Schluss“ ausgeführt
  if (fileReader != null) {
    try {
       fileReader.close();
    } catch (IOException e) {
      System.out.println(“Benachrichtigen Sie“
                              + “den Systemadministrator“);
    }
  }
}
```

Exceptions

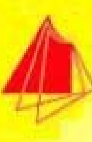
- Return in try-catch-finally
  - Try: return 1 wird ausgeführt (aber Methode nicht verlassen)
  - finally wird ausgeführt mit dem Grund „ Methode zu verlassen und 1 zurückzugeben“
  - Finally: return 2 wird ausgeführt, der Grund hat sich zu „Methode verlassen und 2 zurückgeben“ geändert
  - Danach wird 2 zurückgegeben

```
public int macheWas() {
  try {
    return 1;  // danach wird finally ausgeführt
  } finally {
    return 2;  // es wird immer 2 zurückgegeben
  }
}
```

Exceptions

- Return in try-catch-finally
  - Try: return a++ wird ausgeführt (aber Methode nicht verlassen)
  - finally wird ausgeführt mit dem Grund „ Methode zu verlassen und 1 zurückzugeben“
  - Finally: return 2 wird ausgeführt, der Grund hat sich zu „Methode verlassen und 2 zurückgeben“ geändert
  - Danach wird 2 zurückgegeben

```
public int macheWas() {
  int a = 1;
  try {
    return a++; // a um 1 erhöhen wird ausgeführt
  } finally {
    return a;  // es wird immer 2 zurückgegeben
  }
}
```



Exceptions

```
public int macheWas(String s) {
  int a = 1;
  try {
    s.charAt(0);
    return a++; // a um 1 erhöhen wird ausgeführt
  } finally {
    return a;  // es wird 2 zurückgegeben, falls s != null
               // bei s == null wird Methode mit dem Grund
               // „werfe NullPointerException“ betreten
               // 1 wird zurückgegeben (Grund ändert sich)
  }
}
```

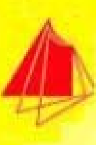
```
public int macheWas(String s) {
  int a = 1;
  try {
    s.charAt(0);
    return a++; // a um 1 erhöhen wird ausgeführt
  } finally {
    // was passiert, wenn s == null war ?
  }
}
```

was passiert hier, wenn s == null war ?

Exceptions

- Finally wird betreten mit dem Grund „Methode mit NullPointerException verlassen“
- Der Grund ändert sich nicht
- Es wird eine NullPointerException geworfen

```
public int macheWas(String s) {
  int a = 1;
  try {
    s.charAt(0);
    return a++; // a um 1 erhöhen wird ausgeführt
  } finally {
    // wenn s == null war, wird NullPointerException geworfen
    // ansonsten wird 2 zurückgegeben
  }
}
```



Exceptions

```
public int macheWas(String s1, String s2) {
  int a = 1;
  try {
    s1.charAt(0);
    return a++;
  } catch (NullPointerException e) {
    s2.charAt(0);
    return a = a + 2;
  } finally {
    if (s1 == null && s2 == null) {
      return a + 4;
    }
  }
}
```

s1	s2	Rück gabe
“a“	“b“	
null	“b“	
“a“	null	
null	null	

Exceptions

```
public int macheWas(String s1, String s2) {
   int a = 1;
   try {
      s1.charAt(0);
      return a++;
   } catch (NullPointerException e) {
      s2.charAt(0);
      return a = a + 2;
   } finally {
      if (s1 == null && s2 == null) {
         return a + 4;
      }
   }
}
```

s1	s2	Rückgabe
“a“	“b“	1
null	“b“	3
“a“	null	1
null	null	5

Exceptions

```
public int macheWas(String s1, String s2) {
   int a = 1;
   try {
      s1.charAt(0);
      return a++;
   } catch (NullPointerException e) {
      s2.charAt(0);
      return a = a + 2;
   } finally {
      return a + 4;
   }
}
```

s1	s2	Rück gabe
“a“	“b“	
null	“b“	
“a“	null	
null	null	

Exceptions

```
public int macheWas(String s1, String s2) {
   int a = 1;
   try {
      s1.charAt(0);
      return a++;
   } catch (NullPointerException e) {
      s2.charAt(0);
      return a = a + 2;
   } finally {
      return a + 4;
   }
}
```

s1	s2	Rück gabe
“a“	“b“	6
null	“b“	7
“a“	null	6
null	null	5

Exceptions

- Checked Exceptions
  - Compiler prüft ob checked Exception gefangen oder mit throw deklariert wurde
- Sinn dieses Unterschieds
  - Fehler, die mit „Exception“ angezeigt werden, können meist nur mit Hilfe des Benutzers gelöst werden.
  - Z.B. „Datei nicht gefunden“: Benutzer muss andere Datei auswählen
  - Anderen Exception-Typen zeigen normalerweise Programmierfehler an: Der Benutzer kann hier nicht helfen - sie zu fangen erhöht nicht die Korrektheit des Programms

Inhalt

- Fehlerbehandlung ohne Exceptions
- Exceptions
  - unchecked exceptions
  - checked exceptions
- Dateien

Dateien

- Bisher
  - FileReader / BufferedReader, um Textdateien einzulesen
- Es wird default-Textcodierung angenommen
  - Hängt von Betriebssystem und zugehöriges Java ab
- Codierung kann auch geändert werden
  - Z.B. ISO Latin-1 unter Windows einlesen
  - Oder UTF-8 unter Unix
  - Angabe nicht bei FileReader möglich, sondern bei Oberklasse InputStreamReader
- Notwendig, wenn Textdaten von anderen Systemen verarbeitet werden sollen
- Java besitzt Vielzahl von Codetabellen zur Konvertierung
  - Intern immer 16-Bit Unicode
  - Existierende Codetabellen hängen von konkret verwendetem Java Interpreter ab

Dateien

- Immer unterstützte Textcodierungen
  - Vom Betriebssystem verwendete
  - Sowie folgende (aus der Javadoc)

Name	Beschreibung
US-ASCII	Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set
ISO-8859-1	ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
UTF-8	Eight-bit UCS Transformation Format
UTF-16BE	Sixteen-bit UCS Transformation Format, big-endian byte order
UTF-16LE	Sixteen-bit UCS Transformation Format, little-endian byte order
UTF-16	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

Dateien

- In Eclipse
  - Textformat pro Datei, Projekt, generell einstellbar
  - Windowscodierung (XP): Cp1252

Properties for info1

type filter te

Info
Builders
Java Build Path
Java Code Style
Java Compiler
Javadoc Location
Project Reference

Info

Path: /info1
Type: Project
Location: D:\vorlesungen\ws_06_07\info1\java\info1

Last modified: 15. Januar 2007 09:23:04

Text file encoding
Inherited from container (Cp1252)
Other: Cp1252

ISO-8859-1
US-ASCII
UTF-16
UTF-16BE
UTF-16LE

Restore Defaults | Apply

OK | Cancel

FileReader
FileReader(File file)
FileReader(String name)

Textcodierung kann hier nicht angegeben werden

InputStreamReader
FileReader(InputStream inputStream,
String charsetName)
readLine() : String

Braucht einen InputStream, kein Reader, File, …

throws UnsupportedEncodingException

Reader <<abstract>>
read() : int
read(char[] cbuf) : int

FileInputStream
FileInputStream(File file)
FileInputStream(String name)

Dateien

- Reader, Writer
  - Für Lesen / Schreiben zeichenbasierter Daten
  - Grundlegende Einheit: char
- InputStream, OutputStream
  - Für Lesen / Schreiben irgendwelcher Daten
  - Grundlegende Einheit: byte
- InputStreamReader, OutputStreamWriter
  - Brücke zwischen beidem
  - Zuständig für Konvertierung von bytes zu Zeichen
  - Information für Textcodierung ist dazu notwendig

Dateien

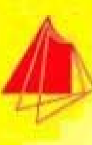
- Lese Textdatei unter Windows (Cp1252), schreibe sie als UTF-8 in eine neue Datei
  - eingabe.txt
  - ausgabe.txt
- Erzeuge
  - Reader für Datei „eingabe.txt“
  - Writer für Datei „ausgabe.txt“
- Methode
  - Um Daten zu kopieren
- Dateien schließen
  - close muss **jeweils** in jedem Fehlerfall aufgerufen werden

Dateien

```
File eingabeDatei = new File("eingabe.txt");
File ausgabeDatei = new File("ausgabe.txt");
Reader reader = null;
Writer writer = null;

try {
  reader = new InputStreamReader(
            new FileInputStream(eingabeDatei),
            "Cp1252" // könnte weggelassen werden (Default)
           );
  writer = new BufferedWriter(
            new OutputStreamWriter(
              new FileOutputStream(ausgabeDatei),
                "UTF-8“
            )
           )

  kopiereDaten(reader, writer);
} catch (IOException e) {
  System.out.println(“Hilfe ein Fehler ist aufgetreten“);
} // finally fehlt noch, um Datei zu schliessen
```

Dateien

```
try {
…  // wie zuvor
} catch ( … ) {
…  // wie zuvor
} finally {
  if (reader != null) {
    try {
      reader.close();
    } catch (IOException e) {
      throw e;
    } finally { // falls reader.close() Exception warf
      if (writer != null) {
        try {
          writer.close();
        } catch (IOException e) {
          throw e;
        }
      }
    }
  }
}
```



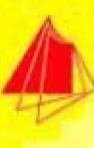
Dateien

- Entwurfshilfen
  - Datei schließen auf selben Ebene wie Datei öffnen
  - Generell: Resourcen dort freigeben, wo sie angefordert wurden
  - Nicht: Datei öffnen, Methode mit dieser Datei aufrufen und Datei in der Methode schließen, sondern danach

```
Reader r = new FileReader(f);

leseDaten(r);

r.close();
```

  - Verarbeitung mit möglichst abstrakten Klassen programmieren
  - Writer, Reader
  - InputStream, OutputStream

```
public void kopiereDaten(Reader reader,
                         Writer writer)
       throws IOException {
  BufferedReader bf =
         new BufferedReader(reader);
  while (reader.ready() ) {
    writer.write( bf.readLine() );
    writer.write("\n");
  }
}
```



Dateien

```
public static void dateiKopieren(File file1, File file2)
         throws IOException  { // keine Fehlerbehandlung in der Methode
  InputStream inputStream = null;
  OutputStream outputStream = null;

  try {
    inputStream = new FileInputStream(file1);
    outputStream = new FileOutputStream(file2);

    while (inputStream.available() > 0) {
      outputStream.write( inputStream.read());
    }
  } finally {
    if (inputStream != null) {
      try {
        inputStream.close();
      } finally {
        if (outputStream != null) {
          outputStream.close();
        }
      }
    }
  }
}
```

- Dateien byteweise kopieren

www.thedailywtf.com [Beitrag vom 16.1.07]

While working with someone else's "utility class,"
David decided to have a little fun and turn on compiler warnings.
It turns out that the compiler is actually sort of good at catching weird abuses in Java. As he put it,

„I turned on some additional compiler warnings,
including one to warn about undocumented empty blocks,
and it complained about the catch block below.
Catching and swallowing exceptions is all too common on this project (as are the completely useless javadoc comments that don't actually document anything),
but this is the first time I've seen someone ...“

Actually, why don't I just show you?

```
/**
  * Insert the method's description here.
  * Creation date: (02/15/00 6:27:04 PM)
  * @param obj java.lang.Object
  * @param buf java.lang.StringBuffer
 */
protected static void doWork(Object obj, StringBuffer buf) {
  try {
     String lastTag = "</" + obj.getClass().getName() + ">";
     int position = buf.toString().lastIndexOf( lastTag );
     String description = getDescription( obj );
     buf.insert( position, description );
   } catch (Exception e) {
        try {
           throw new Exception("Error inserting description!");
        } catch (Exception ex) {
        }
   }
 }
```

I don't really know what this function is for; neither does javadoc, so I don't feel so bad. I guess the point is this: if at first you don't succeed, try-catch again.

Kommentare

I think what we have here should be named "juggling". It throws and then immediately catches it's own exception.

Good lord, please tell me that is the work of more than one brain dead developer. I can wrap my mind around the concept of one person writing throwing the new exception... and then someone else coming along and "fixing" it, but if this was written by a single coder I sure hope he was sniffing glue at the time.

I've seen this one way too many times. I think it stems from a lack of knowledge about the 'throws' keyword.