Propädeutikum
Programmierung in der Bioinformatik

Java – Exceptions und I/O Streams

Thomas Mauermeier

04.12.2018

Ludwig-Maximilians-Universität München

Exceptions

In Java ist so gut wie alles objektorientiert
$\Longrightarrow$ auch Fehlermeldungen (**Exceptions**)

- Fehler tritt in Methode auf
- Methode "wirft" ein Exception-Objekt

Warum als Objekt?

Wir wollen auf Exceptions reagieren können!

```
int[] foo = new int[10];
foo[12] = 42;
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 12
```

Arten von Exceptions

Checked Exceptions

Exceptions die ein (gut geschriebenes) Programm **erwarten** und **abfangen** können soll

⚠ Einziger Exception-Typ der behandelt werden **muss**

Beispiel

Versuch des Öffnens eines Files, der nicht existiert

Runtime Exceptions

Exceptions die in der Regel durch **logische Fehler im Code** entstehen, und daher auch dort logisch behoben werden müssen

Beispiel

Versuchter Zugriff auf Arrayfeld das nicht existiert

Errors

Exceptions die **außerhalb** des Programms entstehen, und daher in der Regel nicht vom Code beeinflusst werden können.

Beispiel

Versuch des Öffnens eines existierenden Files, der wegen Hardwarefehler unlesbar ist

Catch or Specify Requirement

Codeteile die eine **Checked Exception** werfen *können* müssen diese behandeln mittels:

try-catch Blöcke

Behandelt Exception:

```
try {
  // Codeteil der Exception werfen kann
} catch (ExceptionTyp e) {
  // Was tun wenn ExceptionTyp eintritt
} finally {
  // Was immer ausgeführt werden soll
}
```

throws Spezifikation

Gibt Exception weiter:

```
public void foo() throws ExceptionTyp {
  // Inhalt der Methode
  // mit Codeteil der Exception werfen kann
}
```

try-catch Block

try-Block

- Umschließt Codeteil der Exception werfen kann

```
try {
    int[] foo = new int[10];
    foo[12] = 42;
} catch (NegativeArraySizeException e) {
    System.err.println("Array mit negativer Größe!");
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Feld existiert nicht: "
                       + e.getMessage());
} finally {
    System.out.println("Ich werde immer ausgeführt.");
}
```

try-catch Block

catch-Block

- In Klammern: Exception-Typ den man “fangen” will
- Im Block: Was soll passieren wenn diese Exception auftritt?
- Darf weggelassen werden
- Auch möglich: mehrere catch-Blöcke hintereinander

```
try {
    int[] foo = new int[10];
    foo[12] = 42;
} catch (NegativeArraySizeException e) {
    System.err.println("Array mit negativer Größe!");
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Feld existiert nicht: "
                       + e.getMessage());
} finally {
    System.out.println("Ich werde immer ausgeführt.");
}
```

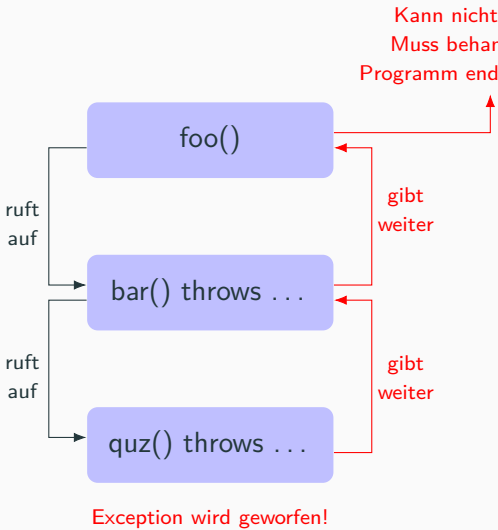
try-catch Block

finally-Block

- Dinge die immer ausgeführt werden sollen (Egal ob Exception auftritt oder nicht)
- Optional, aber sehr praktisch

```
try {
    int[] foo = new int[10];
    foo[12] = 42;
} catch (NegativeArraySizeException e) {
    System.err.println("Array mit negativer Größe!");
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Feld existiert nicht: "
                       + e.getMessage());
} finally {
    System.out.println("Ich werde immer ausgeführt.");
}
```

throws Spezifikation



Eine Methode mit **throws** gibt die Exception den **call stack** nach oben weiter, bis eine Methode die Exception behandelt

```
public void foo() throws ExceptionTyp {
  // Inhalt der Methode
  // mit Codeteil der Exception werfen kann
}
```

Stream I/O

Ein **Stream** repräsentiert ein Objekt aus dem gelesen oder in das geschrieben wird

*Byte*streams

- Schreiben/lesen in Bytes
  $\Rightarrow$ flexibel für verschiedene Daten
- eher low-level I/O
  $\Rightarrow$ vermeiden wenn nicht nötig
- Erben von `InputStream` bzw. `OutputStream`

*Char*streams

- Schreiben/lesen Chars
  $\Rightarrow$ für Textdaten nutzbar
- Spezialisiert auf Text
  $\Rightarrow$ nicht das Rad neu erfinden
- Erben von `Reader` bzw. `Writer`

Streams für Textdateien: FileReader/FileWriter

FileReader – Lesen aus Datei

FileWriter – Schreiben in Datei


```
FileReader fr = null;
FileWriter fw = null;
try {
    fr = new FileReader("haiku.txt");
    fw = new FileWriter("out.txt");
    int c;
    while ((c = fr.read()) != -1) {
        fw.write(c);
    }
} catch (FileNotFoundException e) {
    System.err.println("File existiert nicht!"
} finally {
    if (fr != null) fr.close();
    if (fw != null) fw.close();
}
```

Streams für Textdateien: FileReader/FileWriter

try-catch Block

- Streams werfen immer Checked Exceptions

finally Block

- Streams müssen nach Verwendung geschlossen werden (`.close()`)

```
FileReader fr = null;
FileWriter fw = null;
try {
    fr = new FileReader("haiku.txt");
    fw = new FileWriter("out.txt");
    int c;
    while ((c = fr.read()) != -1) {
        fw.write(c);
    }
} catch (FileNotFoundException e) {
    System.err.println("File existiert nicht!"
} finally {
    if (fr != null) fr.close();
    if (fw != null) fw.close();
}
```

Streams für Textdateien: FileReader/FileWriter

Konstruktor

- `FileReader(String quelle)`
- `FileWriter(String ziel)`

Erstellt "file handle" Objekte über die man aus Quelle (`haiku.txt`) lesen bzw. in Ziel (`out.txt`) schreiben kann

```
FileReader fr = null;
FileWriter fw = null;
try {
    fr = new FileReader("haiku.txt");
    fw = new FileWriter("out.txt");
    int c;
    while ((c = fr.read()) != -1) {
        fw.write(c);
    }
} catch (FileNotFoundException e) {
    System.err.println("File existiert nicht!"
} finally {
    if (fr != null) fr.close();
    if (fw != null) fw.close();
}
```

Streams für Textdateien: FileReader/FileWriter

Lesen mit `fr`

`fr.read()` liest einen Character aus `haiku.txt`, legt ihn in c ab

Beim nächsten `fr.read()` wird der **nächste** Char der Quelle gelesen.

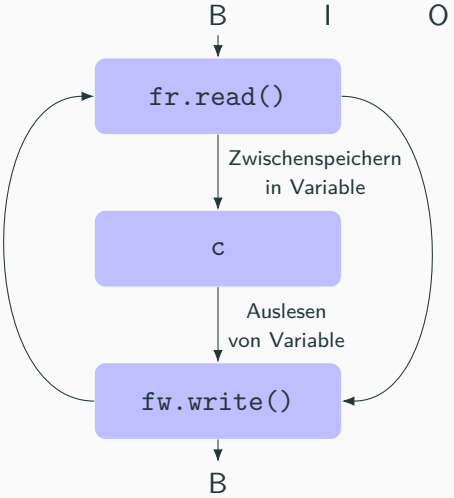
`read()` returned -1 wenn die Quelle "durchgelesen" ist.

Schreiben mit `fw`

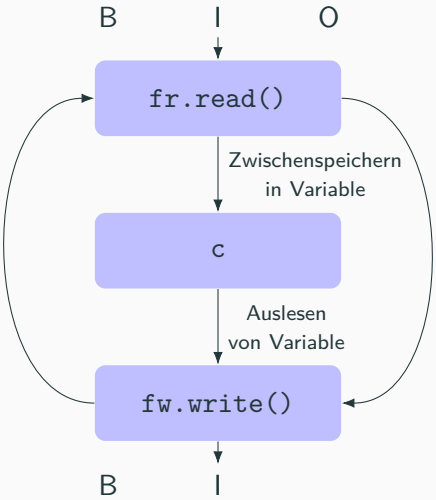
`fw.write(c)` schreibt den im cache abgelegten Char in `out.txt`.

```
FileReader fr = null;
FileWriter fw = null;
try {
    fr = new FileReader("haiku.txt");
    fw = new FileWriter("out.txt");
    int c;
    while ((c = fr.read()) != -1) {
        fw.write(c);
    }
} catch (FileNotFoundException e) {
    System.err.println("File existiert nicht!"
} finally {
    if (fr != null) fr.close();
    if (fw != null) fw.close();
}
```

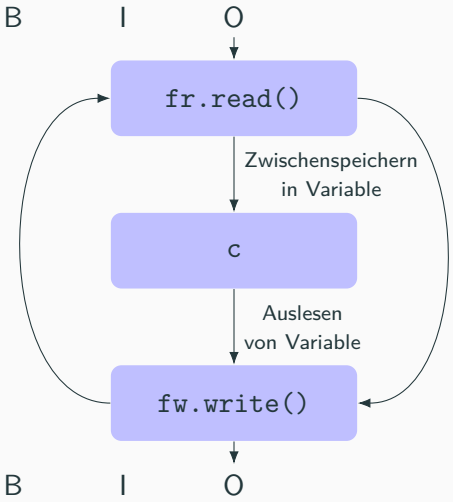
Funktionsweise von FileReader/FileWriter



Funktionsweise von FileReader/FileWriter



Funktionsweise von FileReader/FileWriter



Lesen/Schreiben von Zeilen: BufferedReader/BufferedWriter

Buffered?

- Buffer = Zwischenspeicher
- Jedes mal direkt aus Datei lesen: viele teure Systemoperationen
- Lösung: Größere Menge effizient in Buffer zwischenlagern, erst wieder lesen/schreiben wenn Buffer leer/voll
- $\Rightarrow$ weniger Systemoperationen

```
BufferedReader br = null;
BufferedWriter bw = null;
try {
    br = new BufferedReader(
             new FileReader("haiku.txt"));
    bw = new BufferedWriter(
             new FileWriter("out3.txt"));
    String c;
    while ((c = br.readLine()) != null) {
        bw.write(c);
        bw.newLine();
    }
} finally {
    if (br != null) br.close();
    if (bw != null) bw.close();
}
```

Lesen/Schreiben von Zeilen: BufferedReader/BufferedWriter

BufferedReader

- Konstruktor: `BufferedReader(Reader in)`
- Implementiert `readLine()`: Kann komplette Zeilen einlesen
- Achtung: Liest *nur* die Zeile *ohne* Newline (Zeilenumbruch)

```
BufferedReader br = null;
BufferedWriter bw = null;
try {
    br = new BufferedReader(
             new FileReader("haiku.txt"));
    bw = new BufferedWriter(
             new FileWriter("out3.txt"));
    String c;
    while ((c = br.readLine()) != null) {
        bw.write(c);
        bw.newLine();
    }
} finally {
    if (br != null) br.close();
    if (bw != null) bw.close();
}
```

Lesen/Schreiben von Zeilen: BufferedReader/BufferedWriter

BufferedWriter

- Konstruktor: `BufferedWriter(Writer out)`
- Analog zu `BufferedReader` möglich ganze Strings zu schreiben
- `newLine()` schreibt einen Zeilenumbruch

```
BufferedReader br = null;
BufferedWriter bw = null;
try {
    br = new BufferedReader(
             new FileReader("haiku.txt"));
    bw = new BufferedWriter(
             new FileWriter("out3.txt"));
    String c;
    while ((c = br.readLine()) != null) {
        bw.write(c);
        bw.newLine();
    }
} finally {
    if (br != null) br.close();
    if (bw != null) bw.close();
}
```

try-with-resources Statement

```
BufferedReader br = null;
BufferedWriter bw = null;
try {
    br = new BufferedReader(
             new FileReader("haiku.txt"));
    bw = new BufferedWriter(
             new FileWriter("out.txt"));
    String c;
    while ((c = br.readLine()) != null) {
        bw.write(c);
        bw.newLine();
    }
} finally {
    if (br != null) br.close();
    if (bw != null) bw.close();
}
```

```
try (
    BufferedReader br =
      new BufferedReader(
        new FileReader("haiku.txt"));
    BufferedWriter bw =
      new BufferedWriter(
        new FileWriter("out.txt"))
) {
    String c;
    while ((c = br.readLine()) != null) {
        bw.write(c);
        bw.newLine();
    }
}
```

try-with-resources Statement

try-with-resources

- Gilt für Ressourcen, also Objekte die geschlossen werden müssen
- Deklaration der Streams in Klammern nach dem try
- finally-Block in dem `close()` ausgeführt wird erübrigt sich

```
try (
    BufferedReader br =
      new BufferedReader(
        new FileReader("haiku.txt"));
    BufferedWriter bw =
      new BufferedWriter(
        new FileWriter("out.txt"))
) {
    String c;
    while ((c = br.readLine()) != null) {
        bw.write(c);
        bw.newLine();
    }
}
```