VORSTELLUNG + ORGANISATION

Programmieren Teil 3
Bernhard Fuchs

2025-03-26

Bernhard Fuchs

- Ausbildung:
  - 2021: Master „Information and Computer Engineering“ auf TU Graz
  - Komme aus Graz
  - Researcher and Lecturer @ Campus02

Anwesenheit

LERNZIELE

- File Input / Output
  - Zeichen-orientiert
  - Byte-orientiert
- Exceptions (Error Handling)
- Multithreading
- Netzwerk I/O (Sockets)
  - UDP/TCP
  - Client/Server

LERNZIELE

Lehrinhalte und Lernziele

Lehrinhalte	Grobziele Nach positiver Absolvierung der Lehrveranstaltung sind die Studierenden in der Lage ...	Feinziele
Die Studierenden sollen die Grundlagen von Interaktionen einer Software mit ihrer Umwelt verstehen und anwenden können	Einlesen und Ausgeben von Daten mit lokalen Dateien	
	Exception-Handling kennenlernen und verwenden, eigene Exception-Klassen erstellen	
	Kommunikation über Netzwerke mittels Socket-IO	
	Unterschiede zwischen Verbindungs- und Paket-Orientierung	
	Konzipierung von einfachen Netzwerkprotokollen	
	Anwendung von etablierten Kommunikations-Protokollen wie ZeroMQ, MessagePack und ProtoBuf	
	Anwendungen von Multithreading	
	Verstehen der Herausforderungen und Problemfälle von Software mit mehreren Ablaufsträngen	
	Entwicklung einfacher Server für TCP und UDP mit Multithreading	

https://interface.campus02.at/lv/detail?type=all&lang=de&uniqueUuid=816735ee-16a8-48c1-89fa-5aa73a7aeeee;dac5c0f5-4884-4f6e-be31-453a58eed22d;1

Termine

0004VD1005 PROGRAMMIEREN 3 (51UE IL, SS 2024/25)

Gruppe

Tag	Datum	von	bis	Ort	Ereignis	Termintyp	Lerneinheit	Vortragende*r	Anmerkung
Standardgruppe									
Mi	26.03.2025	08:15	16:00	CZ004	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Mo	31.03.2025	08:15	11:45	CZ004	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Do	03.04.2025	08:15	11:45	CC205	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Do	03.04.2025	12:30	16:00	CC205	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Fr	04.04.2025	08:15	12:15	CZ004	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	Unternehmertag um 11:00 Uhr
Di	08.04.2025	08:15	13:30	CZ004	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Di	29.04.2025	08:15	13:00	CZ004	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Mo	12.05.2025	08:15	16:00	CZ004	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Di	13.05.2025	12:30	16:00	CZ004	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Do	15.05.2025	08:15	16:00	CZ004	Abhaltung	fix		Fuchs, Bernhard, Dipl.-Ing.	
Fr	27.06.2025	08:15	11:45	CZ004	Prüfungstermin	fix			Teilergebnisse: "Klausurarbeit"

UNTERLAGEN - SIEHE MOODLE

- Folien
- Übungsbeispiele (Angaben und Lösungen)
- Java ist eine Insel
  - https://openbook.rheinwerk-verlag.de/javainsel/
- Alte Klausuren

Benotung

Leistungsbeurteilung der Lehrveranstaltungen

Die Leistungsbeurteilung erfolgt auf Lehrveranstaltungsebene	Notenschlüssel* sehr gut (1), gut (2), befriedigend (3), genügend (...

Teilleistung	Gewichtung
Klausurarbeit (schriftlich oder am PC) | Präsenzunterricht	100 % | Mindesterfordernis | Prüfungsgespräch 3. Antritt
Total	100 %

https://interface.campus02.at/lv/detail?type=all&lang=de&uniqueUuid=816735ee-16a8-48c1-89fa-5aa73a7aeeee;dac5c0f5-4884-4f6e-be31-453a58eed22d;1

BENOTUNG

- 30 % theoretische Prüfung in Moodle
- 70 % praktische Programmierprüfung
  - Mit IntelliJ 
  - Hauptklausur: 27.06.2025, 08:15-11:45

File Input Output (I/O)

Programmieren Teil 3
Bernhard Fuchs

2025-03-26

LERNZIELE DIESER FOLIEN

- Auf Dateien zugreifen können
- Die Funktionsweise von Streams (Binär) und Reader (Text) erklären können
- Den Unterschied zwischen Streams und Reader beschreiben können
- Streams und Reader zum Zugriff auf externe Daten einsetzen können
- Verschiedene Dekoratoren kennenlernen, richtig auswählen und einsetzen
- Dekoratoren für Streams und Reader in Hinblick auf eine gegebene Problemstellung auswählen können
- Vordefinierte Dekoratoren für Streams und Reader in zur Lösung gegebener Problemstellungen einsetzen können

LERNHILFEN

- https://www.w3schools.com/java/java_files.asp
- https://www.javatpoint.com/java-file-class
- https://www.decodejava.com/java-bytestream-classes.htm

Input / Output

INPUT / OUTPUT

- Lesen und Schreiben von Daten
  - Quelle? – Woher?
  - Ziel? – Wohin?

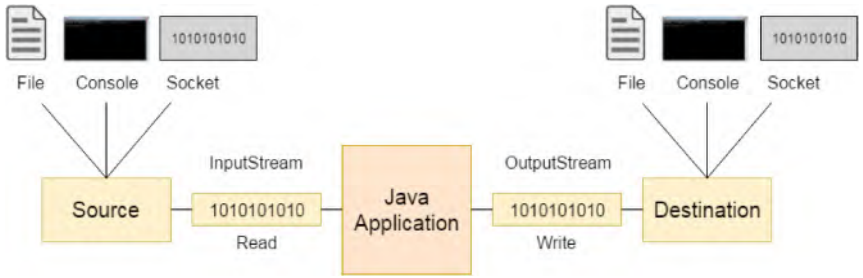
INPUT / OUTPUT

- Lesen und Schreiben von Daten
  - Quelle? – Woher?

INPUT / OUTPUT

- Lesen und Schreiben von Daten
  - Quelle? – Woher?
    - aus Dateien auf der Festplatte / USB-Stick
    - von einem Server im Netzwerk
    - aus dem Arbeitsspeicher
    - aus einer Datenbank
    - …

INPUT / OUTPUT



INPUT / OUTPUT

- Was wird gelesen bzw. geschrieben?
  - Binärdaten
    - Bytes bzw. „Zahlen“
  - Text
    - Characters bzw. Zeichen

BEISPIEL 1

- Ansicht einer Datei im Hex-Editor:

Address Area	Hexadecimal Area																Character Area
00000000	FF	D8	FF	E0	00	10	4A	46	49	46	00	01	01	01	00	60	╪ α..JFIF.....`
00000010	00	60	00	00	FF	E1	11	04	45	78	69	66	00	00	4D	4D	.`.. ß..Exif..MM
00000020	00	2A	00	00	00	08	00	04	01	3B	00	02	00	00	00	13	.*.......;......
00000030	00	00	08	4A	87	69	00	04	00	00	00	01	00	00	08	5E	...Jçi.........^
00000040	9C	9D	00	01	00	00	00	26	00	00	10	D6	EA	1C	00	07	£¥.....&...╔Ω...
00000050	00	00	08	0C	00	00	00	3E	00	00	00	00	1C	EA	00	00	.......>.....Ω..
00000060	00	08	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

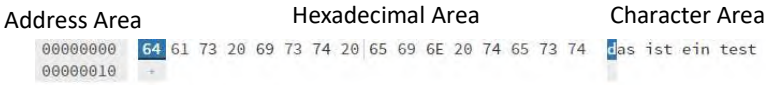
z.B.: https://hexed.it/ oder Hex-Editor Plugin für Notepad++

BEISPIEL 1

- Worum handelt es sich?
  - Um Binärdaten
  - Konkret um ein JPG-Bild
- Was ist an dieser Datei von Interesse?
  - Die Zahlenwerte

BEISPIEL 2

- Ansicht einer **anderer** Datei im Hex-Editor:



z.B.: https://hexed.it/ oder Hex-Editor Plugin für Notepad++

BEISPIEL 2

- Worum handelt es sich?
  - Um Text
  - Konkret um „das ist ein test“
- Was ist an dieser Datei von Interesse?
  - Die Zeichen aus denen der Text besteht

BEISPIEL 2

- Zusammenfassung
  - Dateien können **Binärdaten** enthalten
    - Interessant sind **Zahlen** bzw. **Bytes**
  - Dateien können **Text** beinhalten
    - Interessant sind **Zeichen** bzw. Characters
    - Benötigt wird das **Charset**
      - ASCII, UTF-8, UTF-16, …

DATENQUELLEN

- Wo können Daten herkommen, so dass Java damit arbeiten kann?

DATENQUELLEN

- Konsole
  - System.in
  - System.out
- Dateien
  - File
    - Stream-Klassen
  - Netzwerk
    - Spätere VO

File

KLASSE: **FILE**

- Bildet **plattformunabhängig** eine Datei oder ein Verzeichnis ab
- Stellt Methoden zum:
  - Auslesen von Eigenschaften
  - Prüfen von Zugriffberechtigungen
  - Verwalten von Dateien

  zur Verfügung

KLASSE: FILE CONSTRUCTOR

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

KLASSE: FILE USEFUL METHODS (1/2)

Modifier and Type	Method	Description
static File	createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.

KLASSE: FILE USEFUL METHODS (2/2)

String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

FILE: UE1

- Später auf Moodle verfügbar

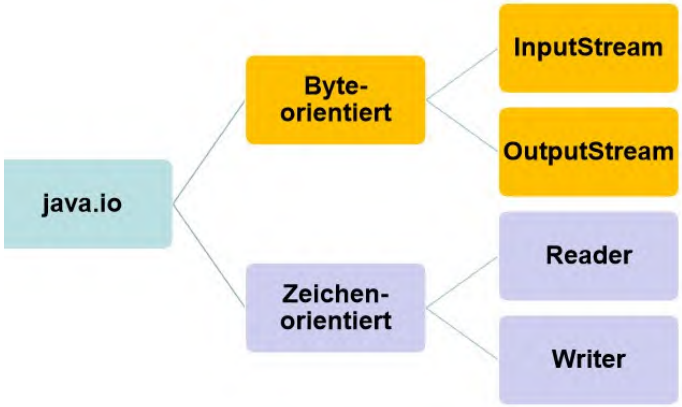
FILE INPUT OUTPUT (I/O) TEIL A: TEXT

Programmieren 3
Bernhard Fuchs

File Input / Output

I/O IN JAVA

Package: java.io



I/O IN JAVA

	Binär	Text
Lesen	InputStream	Reader
Schreiben	OutputStream	Writer

- Ursprünglich gab es nur byteorientierte Streams
  - Byte ist die kleinste Einheit, auf die zugegriffen werden kann

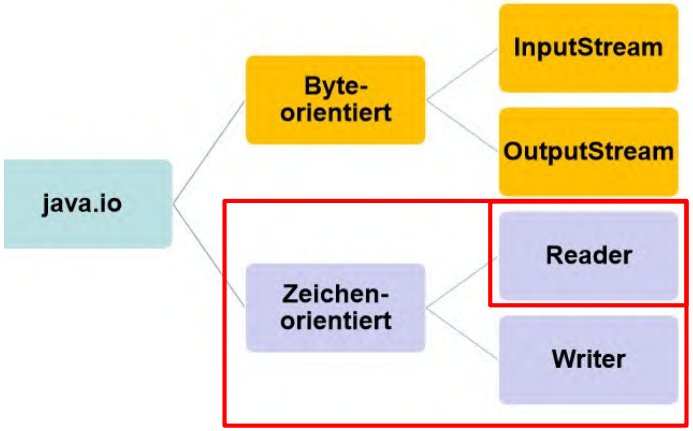
TEXT INPUT

TEXT INPUT

- java.io.Reader
  - ist sehr Low-Level
  - kennt keine Zeilen
  - ist eine abstrakte Klasse
  - hat verschiedene konkrete Subklassen
    - Verschiedene Datenquellen
      - FileReader, StringReader, InputStreamReader
    - Effektivieren Zugriff
      - BufferedReader

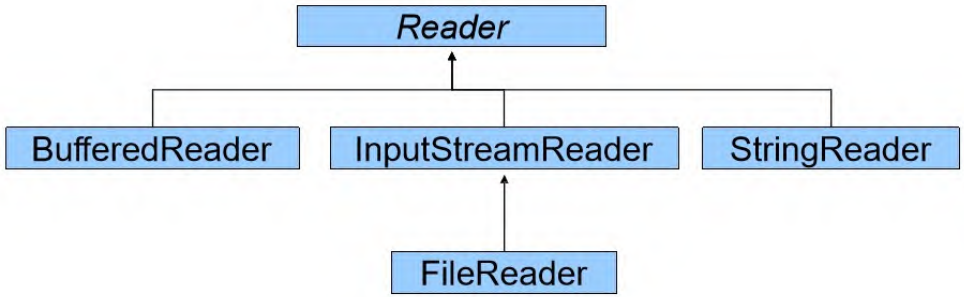
TEXT INPUT

Package: java.io



TEXT INPUT

- Klassen-Hierarchie



TEXT INPUT

Klasse	Beschreibung
Reader	**Abstrakte** Basisklasse zum Lesen von zeichenorientierten Strömen
BufferedReader	Ein Puffer wird für die Leseoperation verwendet
InputStreamReader	Bietet die Möglichkeit einen byteorientieren Stream und einen zeichenorientieren Stream zu koppeln.
FileReader	Leitet von InputStreamReader ab und greift direkt auf Textdateien zu.
StringReader	Mit dieser Klasse kann auf einen String, wie auf einen Stream zugegriffen werden.

BUFFEREDREADER

- Mit dem **BufferedReader** kann auf ganze Zeilen zugegriffen werden. Anstatt auf einzelne Zeichen.

Methoden	Beschreibung
void close()	Schließt den Stream
int read()	Ließt ein Zeichen
String readLine()	Ließt eine Zeile aus dem Stream

TEXT INPUT – LIVE-DEMO

- Live-Demo Beispiel wird im nachhinein auf Moodle zu finden sein.
  - Muss nicht mitgeschrieben werden.
    - Demo: UE-7
    - Weitere Beispiele in Package: textinput

TEXT INPUT BEISPIEL (1/5)

```
File file = new File( pathname: "C:\\campus02\\test.txt");
```

- File öffnet die Datei.

TEXT INPUT BEISPIEL (2/5)

```
File file = new File( pathname: "C:\\campus02\\test.txt");
FileReader fileReader = new FileReader((file));
```

- Liefert einen textorientierten Stream der Datei zurück.

TEXT INPUT BEISPIEL (3/5)

```
File file = new File( pathname: "C:\\campus02\\test.txt");
FileReader fileReader = new FileReader((file));
BufferedReader bufferedReader = new BufferedReader(fileReader);
```

- Legt einen **BufferedReader** über den FileReader. Somit kann komfortabler auf die Datei zugegriffen werden.

TEXT INPUT BEISPIEL (4/5)

```
File file = new File( pathname: "C:\\campus02\\test.txt");
FileReader fileReader = new FileReader((file));
BufferedReader bufferedReader = new BufferedReader(fileReader);

String line;
while ((line = bufferedReader.readLine()) != null) {
    System.out.println(line);
}
```

- Zeile für Zeile wird eingelesen. Ist das Dateiende erreicht, so wird **null** zurückgeliefert.

TEXT INPUT BEISPIEL (5/5)

```
File file = new File( pathname: "C:\\campus02\\test.txt");
FileReader fileReader = new FileReader((file));
BufferedReader bufferedReader = new BufferedReader(fileReader);

String line;
while ((line = bufferedReader.readLine()) != null) {
    System.out.println(line);
}

bufferedReader.close();
```

- Datei wird wieder geschlossen.

TEXT INPUT – **ÜBUNG-7**

- Schreiben Sie ein Programm, welches eine Textdatei mittels BufferedReader bis zum Ende ließt und auf die Konsole ausgibt.

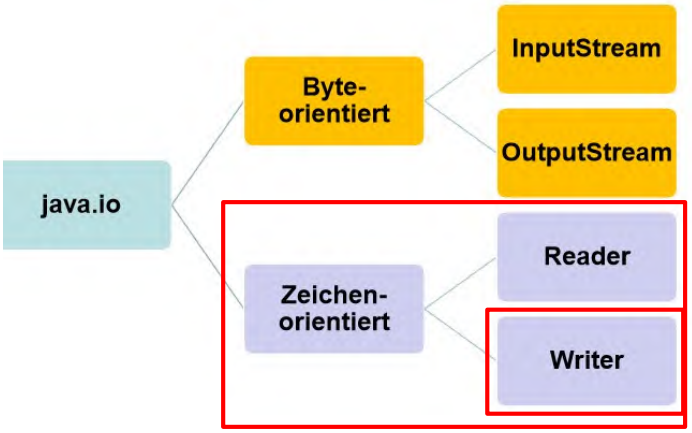
TEXT INPUT – **ÜBUNG-8**

- Schreiben Sie ein Programm, das zeilenweise Tastatureingaben auf die Konsole schreibt, bis das Wort „STOP“ eingegeben wird.
  - Verwenden Sie dazu den InputStream System.in
  - Verwenden Sie weiters die Klassen:
    - InputStreamReader und BufferedReader

TEXT OUTPUT

TEXT INPUT

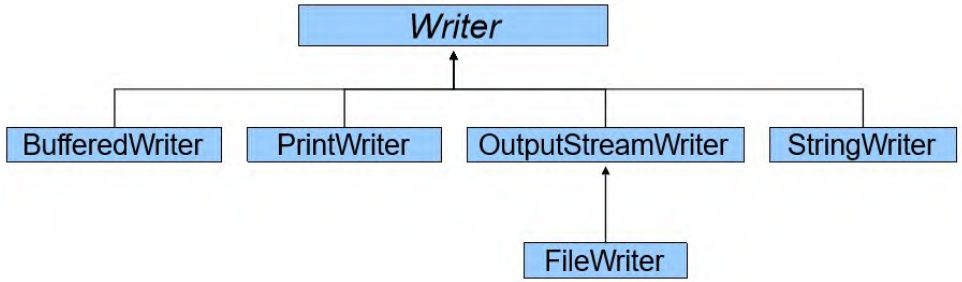
Package: java.io



TEXT OUTPUT

- java.io.Writer
  - Ist wiederum sehr Low-Level
    - Kennt keine print-Methoden
  - Ist eine abstrakte Klasse
  - Hat verschiedene konkrete Subklassen für:
    - Verschiedene Datensenken:
      - FileWriter, StringWriter, OutputStreamWriter
    - Effektiveren Zugriff:
      - BufferedWriter, PrintWriter

TEXT OUTPUT



TEXT OUTPUT

Klasse	Beschreibung
Writer	Basisklasse zum Schreiben von zeichenorientierten Strömen
BufferedWriter	Ein Puffer wird für die Schreiboperationen verwendet
OutputStreamWriter	Bietet die Möglichkeit einen byteorientieren Stream in einen zeichenorientieren Stream zu koppeln.
FileWriter	Leitet von OutputStreamWriter ab und schreibt direkt in Dateien.
StringWriter	Mit dieser Klasse kann auf einen String, wie mit einem Stream geschrieben werden
PrintWriter	Bietet Methoden um einfache Datentypen im Text auszugeben.

PRINTWRITER

- Stellt einzelne Methoden zur Verfügung, mit denen einfache Datentypten entsprechend formatiert werden können.

Method Summary	
PrintWriter	**format**(Locale l, String format, Object... args) Writes a formatted string to this writer using the specified format string and arguments.
PrintWriter	**format**(String format, Object... args) Writes a formatted string to this writer using the specified format string and arguments.
void	**print**(boolean b) Print a boolean value.
void	**print**(char c) Print a character.
void	**print**(char[] s) Print an array of characters.
void	**print**(double d) Print a double-precision floating-point number.

TEXT OUTPUT – LIVE- DEMO

- Live-Demo Beispiel wird im nachhinein auf GitHub zu finden sein.
  - Muss nicht mitgeschrieben werden.
    - Demo: UE-9
    - Weitere Beispiele in Packet: textoutput

TEXT OUTPUT BEISPIEL (1/5)

```
File f = new File( pathname: "campus02-test.txt");
```

- File öffnet die Datei.

TEXT OUTPUT BEISPIEL (2/5)

```
FileWriter fileWriter = new FileWriter(f);
PrintWriter printWriter = new PrintWriter(fileWriter);
```

- Geöffnete Datei wird in den PrintWriter gekapselt.

TEXT OUTPUT BEISPIEL (3/5)

```
printWriter.println("FirstLn");
printWriter.println("SecondLn");
```

- Mittels **println(…)** werden zwei Zeilen geschrieben. Jede Zeile wir richtig abgeschlosen.

TEXT OUTPUT BEISPIEL (4/5)

```
printWriter.flush();
```

- **Flush()** löst das Schreiben aus.

TEXT OUTPUT BEISPIEL (5/5)

```
printWriter.close();
```

- Close() gibt die Datei wieder frei

TEXT OUTPUT – ÜBUNG-9

- Schreiben Sie ein Programm, welches eine Textdatei mittels Filewriter und Printwriter erstellt.

TEXT OUTPUT – ÜBUNG-10

- Schreiben Sie ein Programm, das Ihre Noten aus allen Fächern im ersten Semester von der Konsole einliest und anschließend als „.txt“ Datei speichert.
  - Beispiel:
    - PR1: 1
    - Englisch: 1
- Die Eingabe endet, wenn das Wort „STOP“ eingegeben wird.
- Verwenden Sie die Klassen *BufferedReader, InputStreamReader und FileWriter*

TEXT IO ÜBUNG-11

- Schreiben Sie eine Klasse **Product** zur Abbildung von Produkten mit folgenden Attributen.
  - String ProductName
  - Double Price
  - String ProductCategory
- Schreiben Sie eine Klasse **ProductManager** zum Verwalten von Produkten in einer privaten Liste und implementieren Sie folgende Methoden
  - public void add(Product p) ➔ fügt ein Produkt hinzu
  - public void saveToFile(String path) ➔ speichert die Produkte in der übergebenen Datei. - Realisieren Sie dies mittels BufferedWriter und Filewriter.
  - public void readFromFile(String path) ➔ laden Sie den Text von der übergebenen Datei und geben Sie den Text auf der Konsole aus.

FILE INPUT OUTPUT (I/O) TEIL B: BINARY

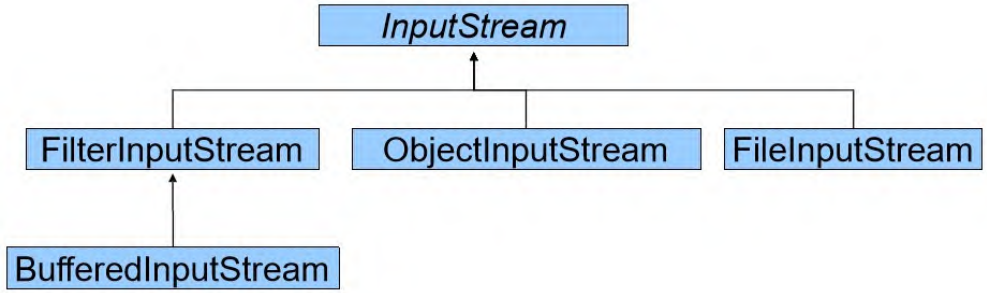
Programmieren 3
Bernhard Fuchs

JAVA BINARY INPUT

- Java.io.InputStream
  - Ist sehr Low-Level
  - Ist **Closeable**
  - Ist eine **abstrakte** Klasse
  - Hat verschiedene konkrete Subklassen
    - Für verschiedene Datenquellen
      - FileInputStream, ByteArrayInputStream, AudioInputStream
    - Für effektiveren Zugriff
      - BufferedInputStream

JAVA BINARY INPUT

- Klassenhierarchie



INPUTSTREAM

Klasse	Beschreibung
InputStream	**Abstrakte** Basisklasse zum Lesen für byteorientierte Streams
FileInputStream	Liest aus Dateien
ObjectInputStream	Stellt Methoden zur Verfügung, mit denen „gesamte Objekte“ gelesen werden können
FilterInputStream	Lässt direkt beim Einlesen das Bearbeiten (z.B.: Entschlüsselung von Daten) von Dateien zu und dient als Basisklasse für weitere Stream-Klassen.
BufferedInputStream	Klasse, die über einen optimierten Zugriff (Puffer) auf Dateien verfügt.

BINARY INPUT – LIVE- DEMO

- Übung 12
  - Weitere Beispiele im Paket: binaryinput

BINARY INPUT BEISPIEL (1/4)

```
File file = new File( pathname: "campus02-test.txt");
```

- Repräsentiert eine Datei auf dem Dateisystem.

BINARY INPUT BEISPIEL (2/4)

```
File file = new File( pathname: "campus02-test.txt");
FileInputStream fileInputStream = new FileInputStream(file);
```

- FileInputStream stellt eine Verbindung zur Datei her. Datei wird geöffnet.

BINARY INPUT BEISPIEL (3/4)

```
File file = new File( pathname: "campus02-test.txt");
FileInputStream fileInputStream = new FileInputStream(file);

int byteRead;
while ((byteRead = fileInputStream.read()) != -1) {
    char[] ch = Character.toChars(byteRead);
    System.out.println(ch[0]);
}
```

- read() liefert Byte for Byte aus der Datei. Wenn das Ende erreicht ist, dann liefert die Methode -1 als Ergebnis.
- While-Schleife liest somit Zeichen um Zeichen aus.

BINARY INPUT BEISPIEL (4/4)

```
File file = new File( pathname: "campus02-test.txt");
FileInputStream fileInputStream = new FileInputStream(file);

int byteRead;
while ((byteRead = fileInputStream.read()) != -1) {
    char[] ch = Character.toChars(byteRead);
    System.out.println(ch[0]);
}

fileInputStream.close();
```

- close() gibt die Datei wieder frei.

BINARY INPUT **ÜBUNG 13**

- Schreiben Sie ein Programm, welches von einer Datei alle Zeichen einliest und auf der Konsole ausgibt.
- Zählen Sie die Anzahl der eingelesenen Zeichen und geben Sie die Gesamtanzahl am Ende aus.

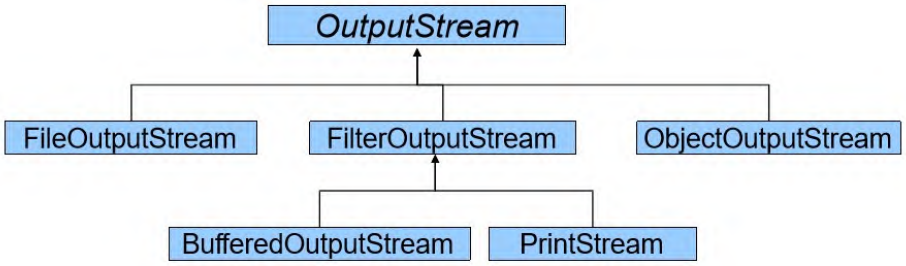
- Solution: ue_13 or ue_13_buffered_reader

BINARY OUTPUT

BINARY OUTPUT

- Java.io.OutputStream
  - Ist wiederum sehr Low-Level
  - Ist Closeable and Flushable
  - Ist eine abstrakte Klasse
  - Hat verschiedene konkrete Subklassen für:
    - Verschiedene Datensenken
      - FileOutputStream, ByteArrayOutputStream
    - Effektiveren Zugriff
      - FilterOutputStream, PrintOutputStream, BufferedOutputStream

BINARY OUTPUT



BINARY OUTPUT

Klasse	Beschreibung
OutputStream	**Abstrakte** Basisklasse zum Schreiben für byteorientierte Streams
FileOutputStream	Schreibt in Dateien
ObjectOutputStream	Stellt Methoden zur Verfügung, mit denen „gesamte Objekte“ geschrieben werden können
FilterOutputStream	Manipuliert direkt beim Schreiben den Output (z.B.: Verschlüsselung) und dient als Basisklasse für weitere Stream-Klassen.
BufferedOutputStream	Klasse, die über einen optimierten Zugriff (Puffer) auf Dateien verfügt.
PrintStream	Stellt Methoden zur zeilenorientierten Ausgabe (print / println) zur Verfügung

BINARY OUTPUT – LIVE-DEMO

- UE 14
  - Demo: ue_14
  - Weitere Beispiele im Paket binaryoutput

BINARY OUTPUT BEISPIEL (1/5)

```
File file = new File( pathname: "test-output.txt");
FileOutputStream fileOutputStream = new FileOutputStream(file);
```

- FileOutputStream öffnet die Datei. Der Konstruktur bietet Einstellungen, ob Datei überschrieben werden darf oder nicht.

BINARY OUTPUT BEISPIEL (2/5)

```
File file = new File( pathname: "test-output.txt");
FileOutputStream fileOutputStream = new FileOutputStream(file);

String outputText = "hello File - first output";
for (char c : outputText.toCharArray()) {
    fileOutputStream.write(c);
}
```

- Ausgabe erstellen und diese Zeichen für Zeichen durchlaufen

BINARY OUTPUT BEISPIEL (3/5)

```
File file = new File( pathname: "test-output.txt");
FileOutputStream fileOutputStream = new FileOutputStream(file);

String outputText = "hello File - first output";
for (char c : outputText.toCharArray()) {
    fileOutputStream.write(c);
}
```

- Zeichen mittels .write(…) in die Datei schreiben.

BINARY OUTPUT BEISPIEL (4/5)

```
File file = new File( pathname: "test-output.txt");
FileOutputStream fileOutputStream = new FileOutputStream(file);

String outputText = "hello File - first output";
for (char c : outputText.toCharArray()) {
    fileOutputStream.write(c);
}
fileOutputStream.flush();
```

- .flush() fürht den tatsächlichen Schreibvorgang durch. Darf NICHT vergessen werden!!

BINARY OUTPUT BEISPIEL (5/5)

```
File file = new File( pathname: "test-output.txt");
FileOutputStream fileOutputStream = new FileOutputStream(file);

String outputText = "hello File - first output";
for (char c : outputText.toCharArray()) {
    fileOutputStream.write(c);
}
fileOutputStream.flush();
fileOutputStream.close();
```

- .close() gibt die Datei wieder frei. Darf ebenso NICHT vergessen werden.

BINARY OUTPUT ÜBUNG 15

- Lesen Sie Zeichen für Zeichen von der Konsole ein und schreiben Sie die einzelnen Bytes in eine Datei.
- Von der Konsole kann mittels System.in.read() ein Zeichen gelesen werden
- Brechen Sie ab, wenn ein ‘x‘ eingegeben wird
- Tipp:

```
char c = (char) System.in.read();
if (c == 'x')
{
        System.out.println("Es kam ein x");
}
```

- Solution: ue_15

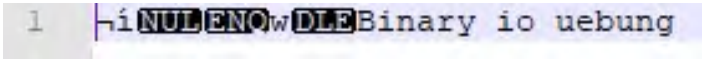
KOMBINIEREN VON KLASSEN

- Java.IO verwendet das Decorator-Pattern an
- Klassen können somit miteinander kombiniert bzw. „dekoriert“ werden.
- Bspw. Nimmt BufferedInputStream ein InputStream-Objekt auf
  - Dieses kann ein FileInputStream
  - Oder auch ein anderer InputStream sein

BINARY IO ÜBUNG

- Schreiben Sie ein Programm, das ein String- Objekt „Hallo Welt“ in eine Datei „object.dat“ serialisiert und anschließend aus dieser wieder ausliest und auf die Konsole schreibt.
- Verwenden Sie die Klassen FileOutputStream und ObjectOutputStream sowie FileInputStream und ObjectInputStream
- Verwenden Sie für das Schreiben die Methode writeObject(…), für das Lesen die Methode readObject(). Beim Lesen müssen Sie das Ergebnis in einen String casten
- Betrachten Sie die Datei in einem Editor (Notepad, Notepad++)
- Solution: Klasse: ue_16

BINARY IO ERGEBNIS



- Steuerzeichen sagen dem Leser, um welches Objekt es sich handelt. Hiermit können auch komplexe / eigene Klassen serialisiert werden.

BINARY IO ÜBUNG

- Schreiben Sie eine Klasse Product zur Abbildung von Produkten mit folgenden Attributen.
  - String ProductName
  - Double Price
  - String ProductCategory
- Schreiben Sie eine Klasse ProductManager zum Verwalten von Produkten in einer privaten Liste und implementieren Sie folgende Methoden
  - public void add(Product p) ➔ fügt ein Produkt hinzu
  - public void save(String path) ➔ speichert die Produkte in der übergebenen Datei. Realisieren Sie dies mittels ObjectOutputSteam. Kombinieren Sie den ObjectOutputStream mit einem FileOutputStream.
  - public void load(String path) ➔ laden Sie die Produkte von der übergebenen Datei
- Lösung: Package: ue17_binary_io

ZEICHENKODIERUNG

ZEICHENORIENTIERTE DATEIEN

- Lesen / Schreiben Unicode Zeichen
- Zeichenlänge hängt von der tatsächlichen Kodierung ab:
  - UTF-8
  - UTF-16
  - Latin-1
  - …

ISO-8859-15

```
00000000: 61e4 6263 6465 6667  a.bcdefg
00000008: 6869 6a6b 6c6d 6e6f  hijklmno
00000010: f670 7172 73df 7475  .pqrs.tu
00000018: fc76 7778 797a 0a41  .vwxyz.A
00000020: c442 4344 4546 4748  .BCDEFGH
00000028: 494a 4b4c 4d4e 4fd6  IJKLMNO.
00000030: 5051 5253 df54 55dc  PQRS.TU.
00000038: 5657 5859 5a0a       VWXYZ.
```

Größe: **62** Bytes

UTF-16

```
00000000: 6100 e400 6200 6300  a...b.c.
00000008: 6400 6500 6600 6700  d.e.f.g.
00000010: 6800 6900 6a00 6b00  h.i.j.k.
00000018: 6c00 6d00 6e00 6f00  l.m.n.o.
00000020: f600 7000 7100 7200  ..p.q.r.
00000028: 7300 df00 7400 7500  s...t.u.
00000030: fc00 7600 7700 7800  ..v.w.x.
00000038: 7900 7a00 0a00 4100  y.z...A.
00000040: c400 4200 4300 4400  ..B.C.D.
00000048: 4500 4600 4700 4800  E.F.G.H.
00000050: 4900 4a00 4b00 4c00  I.J.K.L.
00000058: 4d00 4e00 4f00 d600  M.N.O...
00000060: 5000 5100 5200 5300  P.Q.R.S.
00000068: df00 5400 5500 dc00  ..T.U...
00000070: 5600 5700 5800 5900  V.W.X.Y.
00000078: 5a00 0a00            Z...
```

Größe: **124** Bytes

UTF-8

```
00000000: 61c3 a462 6364 6566  a..bcdef
00000008: 6768 696a 6b6c 6d6e  ghijklmn
00000010: 6fc3 b670 7172 73c3  o..pqrs.
00000018: 9f74 75c3 bc76 7778  .tu..vwx
00000020: 797a 0a41 c384 4243  yz.A..BC
00000028: 4445 4647 4849 4a4b  DEFGHIJK
00000030: 4c4d 4e4f c396 5051  LMNO..PQ
00000038: 5253 c39f 5455 c39c  RS..TU..
00000040: 5657 5859 5a0a       VWXYZ.
```

Größe: **70** Bytes

ZEICHENKODIERUN: UNTERSCHIEDE

- Die letzten drei Beispiele:
  - einmal 62,
  - einmal 124,
  - einmal 70 Bytes beinhalten den gleichen Text
- Der Unterschied liegt in der Kodierung der Zeichen:
  - ein Byte pro Zeichen
  - zwei Bytes pro Zeichen
  - ein Byte für gewisse Zeichen, zwei Bytes für andere

EIN BYTE PRO ZEICHEN

- 256 verschiedene Zeichen darstellbar
- Beispiele:
  - ASCII ISO 8859-15 (auch Latin 1)
  - Windows-1252 (Westeuropäische Sprachen))
  - Windows-1250 (Zentral- und Osteuropa))
  - ...

UNICODE TRANSFORMATION FORMAT

- UTF-16:
  - (meist) zwei Bytes pro Zeichen
  - verwendet für z.B. Java String
- UTF-8:
  - variable Länge
  - ein Byte für ASCII Zeichen
  - bis zu 4 Bytes für andere Zeichen
  - Sehr häufig im WWW verwendet

CODIERTE TEXT DATEIEN LIVE-DEMO

- Live-Demo Beispiel wird im nachhinein auf GitHub zu finden sein.
  - Muss nicht mitgeschrieben werden.
    - Demo: codierte_text_dateien_erstellen
    - Weitere Beispiele im Paket: umlaute

CODIERTE TEXT DATEIEN ÜBUNG

- Erstellen Sie ein Programm, welches die zuvor erstellte Text Datei, welche Umlaute beinhaltet, ausließt.
  - Beim auslesen, soll es möglich sein, die gewünschte Codierung anzugeben.
  - Testen Sie den Konsolen Output mit folgenden Codierungsoptionen:
    - UTF-8
    - ISO_8859_1
  - Welche Unterschiede stellen Sie fest?
  - Lösung: ue_codierte_text_dateien_lesen

IO EXCEPTIONS

IO EXCEPTIONS

- Problem bei I/O
  - Was kann alles schief gehen?
    - ???

IO EXCEPTIONS

- Problem bei I/O
  - Es kann potentiell etwas schief gehen
    - Datei existiert nicht
    - Festplatte voll / defekt
    - Netzwerkverbindung bricht ab
    - …

IO EXCEPTIONS

Exception Summary	
CharConversionException	Base class for character conversion exceptions.
EOFException	Signals that an end of file or end of stream has been reached unexpectedly during input.
FileNotFoundException	Signals that an attempt to open the file denoted by a specified pathname has failed.
InterruptedIOException	Signals that an I/O operation has been interrupted.
InvalidClassException	Thrown when the Serialization runtime detects one of the following problems with a Class.
InvalidObjectException	Indicates that one or more deserialized objects failed validation tests.
IOException	Signals that an I/O exception of some sort has occurred.
NotActiveException	Thrown when serialization or deserialization is not active.
NotSerializableException	Thrown when an instance is required to have a Serializable interface.
ObjectStreamException	Superclass of all exceptions specific to Object Stream classes.
OptionalDataException	Exception indicating the failure of an object read operation due to unread primitive data, or the end of data belonging to a serialized object in the stream.
StreamCorruptedException	Thrown when control information that was read from an object stream violates internal consistency checks.
SyncFailedException	Signals that a sync operation has failed.
UnsupportedEncodingException	The Character Encoding is not supported.
UTFDataFormatException	Signals that a malformed string in modified UTF-8 format has been read in a data input stream or by any class that implements the data input interface.
WriteAbortedException	Signals that one of the ObjectStreamExceptions was thrown during a write operation.

IO EXCEPTIONS – **UEBUNG**

- Versuchen Sie herauszufinden, welche Exceptions in den bisherigen Beispielen geworfen werden können?
- Stellen Sie Ihre Beispiele so um, dass eventuell auftretende Exceptions direkt behandelt und nicht nach außen geworfen werden.
- Beispiele siehe: Paket textinput/exceptionhandling

IO EXCEPTIONS ÜBUNG BEISPIEL

```java
public static void main(String[] args) {

    // Create a new File and check if the file was successfully created.
    // - Also consider File Separation from Operating System

    // Erstellung eines neuen Files
    File file = new File( pathname: "campus02-test.txt");

    // Receive file separator from System. For Windows: "\"
    String fileSeparator = System.getProperty("file.separator");
    System.out.println("File Separator: " + fileSeparator);

    File file2 = new File( pathname: "C:" + fileSeparator + "Users" + fileSeparator + "" +
            "campus02" + fileSeparator + "neuertest1.txt");

    try {
        if (file2.createNewFile()) {
            System.out.println(file2.getAbsolutePath() + " filename: " + file2.getName());
            System.out.println("New File is created");
        } else {
            System.out.println("File already exists");
        }
    } catch (IOException e) {
        // File / Path does not exists or can not be found
        e.printStackTrace();
    }

}
```


EXCEPTIONS

Programmieren 3
Bernhard Fuchs

INHALT

- Konzepte zur Fehlerbehandlung von Java / JVM
  - Errors
  - (un)checked Exceptions
- Exceptions
  - auslösen („werfen“), behandeln fangen
  - Schlüsselwörter: try, catch, finally, throw/s
  - StrackTraces verstehen und interpretieren
  - Exception Hierarchie & Typen
  - Exception Chaining

LERNZIELE

- Gründe / Vorteile für Exception Mechanismus zur Fehlerbehandlung nennen können
- spezifische Exceptions definieren können
- Exceptions im Fehlerfall auslösen sowie geeignet behandeln können
- Unterschied zwischen *checked* und *unchecked* Exceptions erklären und richtig einsetzen

EXCEPTIONS

- Was sind „Exceptions“?
  - Ausnahmesituationen die im normalen bzw . geplanten Programmablauf auftreten können
  - Zustände die verhindern , dass das Programm seiner Aufgabe ungehindert nachkommen kann
  - *salopp formuliert*: Fehler unterschiedlichster Art

EXCEPTIONS

- Bsp. für bereits kennengelernte Exceptions:

```
public class DemoExceptions {

    public static void main(String[] args) {

        int a = 10;
        int b = 0;

        System.out.println(doDivision(a, b));

    }

    public static int doDivision(int i1, int i2) {
        return i1 / i2;
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at exceptions.DemoExceptions.doDivision(DemoExceptions.java:15)
    at exceptions.DemoExceptions.main(DemoExceptions.java:10)
```


EXCEPTIONS

- Bsp. für bereits kennengelernte Exceptions:

```
public class NumbersArray {

    public static void main(String[] args) {

        int[] numbers = {10,20,30,40,50};

        for(int n=0;n <= numbers.length; n++) {
            System.out.println(numbers[n]);
        }

    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at exceptions.NumbersArray.main(NumbersArray.java:10)
```


EXCEPTIONS

- Bsp. für bereits kennengelernte Exceptions:

```
public class NameRegister {

    private List<String> names;

    public void addName(String name) {
        names.add(name);
    }

    public static void main(String[] args) {

        NameRegister nr = new NameRegister();
        nr.addName("Max Mustermann");

    }
}
```

```
Exception in thread "main" java.lang.NullPointerException
    at exceptions.NameRegister.addName(NameRegister.java:10)
    at exceptions.NameRegister.main(NameRegister.java:17)
```

EXCEPTIONS

- Bsp. für bereits kennengelernte Exceptions:
  - Division durch 0
    - ArithmeticException
  - Zugriff auf Null Referenz
    - NullPointerException
  - ungültiger Array Index
    - ArrayIndexOutOfBoundsException

EXCEPTION HANDLING

- Mechanismus der
  - normalen Code von Fehlerbehandlung klar trennt
  - den Umgang mit Ausnahmesituationen sicherstellt
    - auf Fehler reagieren
  - Aufschluss darüber gibt:
    - WAS passiert ist (Art des Fehlers)
    - WO der Fehler aufgetreten ist (Zeile im Code)
    - WARUM der Fehler aufgetreten ist Beschreibung

EXCEPTION HANDLING

- an der Stelle, wo der Fehler auftritt, wird eine Exception ausgelöst **(„werfen“ →„throw“)**
- an der Stelle, wo der Fehler behandelt werden soll, wird die Exception verarbeitet **(„fangen →„catch“)**
- zwischen dem Auftreten <-> Verarbeitung von Ausnahmen können mehrere Methoden liegen die evt. auch „übersprungen werden (siehe *call stack* bzw. stack trace)

EXCEPTION SYNTAX

- Java Language Keywords

keyword	Bedeutung / Funktion
try	**try** Block umschließt kritischen Code
catch	**catch** Block zur Behandlung bestimmter Exceptions im Fehlerfall
finally	**finally** Block für Code der jedenfalls ausgeführt werden muss, egal ob Fehler auftritt oder nicht
throw	**throw** löst eine Exception aus und signalisiert damit Ausnahmesituationen bzw. Fehler
throws	**throws** als Zusatz bei Methodendeklaration zeigt mögliche Exceptions an, die auftreten könnten und vom Aufrufer behandelt od. weiter delegiert werden

EXCEPTION HANDLING

- einfache Fehlerbehandlung

```
try {

    //TRY BLOCK:
    //any code that may throw an exception

} catch(Exception exc) {

    //CATCH BLOCK:
    //in this example the handling
    //only prints the stack trace information
    exc.printStackTrace();

} finally {

    //OPTIONAL FINALLY BLOCK:
    //code to cleanup any resources

}
```


EXCEPTION HANDLING

- Methodendeklaration signalisiert mögliche Exception(s) die auftreten können

```
public void doSomething() throws SomeException,... {

    // any code that may throw an Exception
    // without caring for it but instead
    // delegate the handling to the caller

}
```

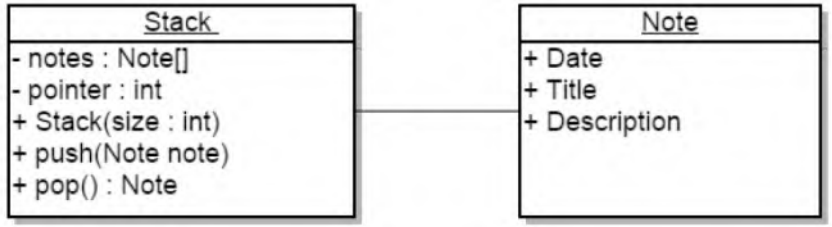

EXCEPTION OBJECTS

- Exceptions sind ebenfalls Java Objekte
- daher auch der *new* Operator beim Erzeugen und Werfen der Ausnahme
  - -> throw new StackEmptyException
- unterstützen die besprochenen OOP Konzepte wie **Vererbung und Polymorphismus**
  - Ableitung z.B . von **java.lang.Exception**
  - erbt davon Methoden wie z.B. **printStackTrace()**

BEISPIEL: STACK

- Implementierung eines einfachen Stack
  - 2 Methoden: **push()** und **pop()**
    - **push()** legt ein Objekt auf den Stapel
    - **pop()** nimmt das oberste Objekt vom Stapel und liefert es zurück
  - maximale Größe des Stacks soll beschränkt sein

BEISPIEL: STACK



BEISPIEL: STACK

- 2 Ausnahmesituationen:
  - push() obwohl Stack voll -> max. Größe erreicht
  - pop() trotz leerem Stack -> keine Elemente mehr
- Umsetzung mittels eigener Exception Types
  - Stack löst Exceptions aus -> throw new …
  - bei push() wenn Stack voll -> StackFullException
  - bei pop() wenn Stack leer -> StackEmptyException

BEISPIEL: STACK

- benutzerdefinierte Exceptions erstellen z.B
  - Ableitung von Basisklasse java.lang.Exception
  - Konstruktor Überladung für unterschiedl. Erzeugung

```
public class StackEmptyException extends Exception {

    public StackEmptyException() {}

    public StackEmptyException(String msg) {
        super(msg);
    }

}
```


EXCEPTION KEYWORDS

- **throw**
  - Auslösen („werfen“) einer Exception im Fehlerfall
  - **WO? An der Stelle des Auftratens der Ausnahme**

```
//in case the underlying array is already full
if(pointer == notes.length-1) {
    throw new StackFullException(“Fehlermeldung…”);
}
```

EXCEPTION KEYWORDS

- **throws**
  - als Teil der Methodendeklaration
  - signalisiert dem Aufrufer potentielle Exceptions vom angegeben Typ die auftreten können

```
public void push(Note note) throws StackFullException {

    //code here that may throw StackFullException

}
```


BEISPIEL: STACK

```
public void push(Note note) throws StackFullException {

    if(pointer == notes.length-1) {
        throw new StackFullException(notes.length);
    }

    notes[++pointer] = note;
}
public Note pop() throws StackEmptyException {

    if(pointer == -1) {
        throw new StackEmptyException("cannot pop stack");
    }
    Note n = notes[pointer];
    notes[pointer--] = null;
    return n;
}
```


EXCEPTION KEYWORDS

- **try & catch**
  - kritischen Code Bereich mit **try** Block einschließen
  - auf potentielle Ausnahmen reagieren bzw . diese geeignet im catch Block behandeln

```
try {
stack.push(...); //may throw StackFullException
}
catch (StackFullException e) {
//handle the exception here
}
```


EXCEPTION KEYWORDS

- **finally**
  - **WICHTIG:** für den Fall das verwendete Resourcen wieder freigegeben werden müssen
    - (z.B. geöffnete Dateien, Netzwerkverbindung, Datenbankverbindung, etc.)
  - **Beispiel:** Methode öffnet File zum Lesen und es kommt zu IOException -> geöffnete Datei muss wieder geschlossen werden

EXCEPTION KEYWORDS

- **finally**
  - sofern gewisse Code Teile jedenfalls ausgeführt werden müssen unabhängig ob Ausnahmen auftreten od. nicht
  - Variante 1: try → catch → finally
  - Variante 2: try → finally

EXCEPTION KEYWORDS

- **finally**

```
try
{
    doSomeThing();
}
catch (SomeException e)
{
    handleException();
}
finally
{
    cleanup();
}
```

```
try
{
    doSomeThing();
}
finally
{
    cleanup();
}
```


BEISPIEL: STACK

```
public static void main(String[] args) {
    int numNotes = 10;
    Stack stack = new Stack(numNotes);
    try {
        stack.push(new Note(LocalDate.now(),"TODO","DESC"));
        System.out.println(stack.pop());
        for(int i=0; i < numNotes; i++) {
            stack.push(new Note(LocalDate.now(),
                    "TODO"+(i+1),"DESC"+(i+1)));
        }
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    } catch (StackEmptyException e) {
        e.printStackTrace();
    } catch (StackFullException e) {
        e.printStackTrace();
    }
}
```


EXCEPTIONS STACKTRACE

```
try {
    //pop from empty stack causes exception
    stack.pop();
} catch (StackEmptyException e) {
    e.printStackTrace();
}
```

→ exceptions.stack.StackEmptyException:
error: cannot pop the empty stack
at exceptions.stack.Stack.pop(Stack.java:27)
at exceptions.stack.StackDemoError.main(StackDemoError.java:14)

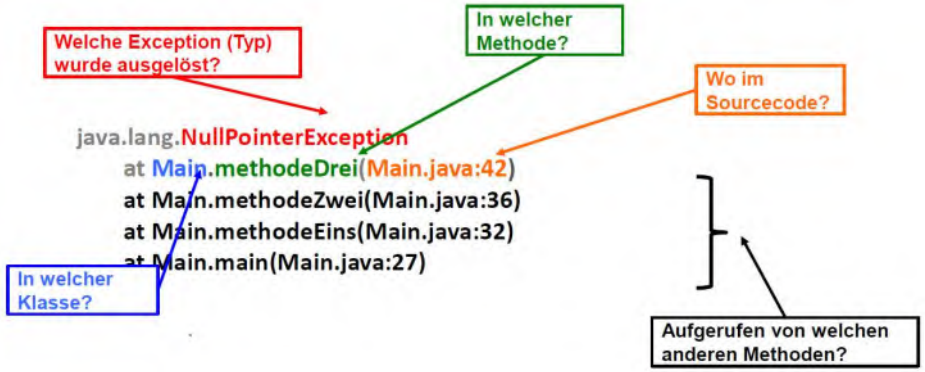
EXCEPTIONS STACKTRACE

```
try {
  //stack's capacity is only e.g. 2 notes
  stack.push(new Note(LocalDate.now(),"TODO1","DESC1"));
  stack.push(new Note(LocalDate.now(),"TODO2","DESC2"));
  //then 3rd push causes exception
  stack.push(new Note(LocalDate.now(),"TODO3","DESC3"));
} catch (StackFullException e) {
  e.printStackTrace();
}
```

```
→ exceptions.stack.StackFullException:
    error: stack is full - max size is: 2
    at exceptions.stack.Stack.push(Stack.java:16)
    at exceptions.stack.StackDemoError.main(StackDemoError.java:23)
```


EXCEPTIONS STACKTRACE

- StackTrace -> exc.printStackTrace



EXCEPTION KEYWORDS

- **WAS** … ist passiert
  - Art des Fehlers
  - aus Klasse der Exception ableitbar
    - StackFullException
    - StackEmptyException
    - etc.

EXCEPTION KEYWORDS

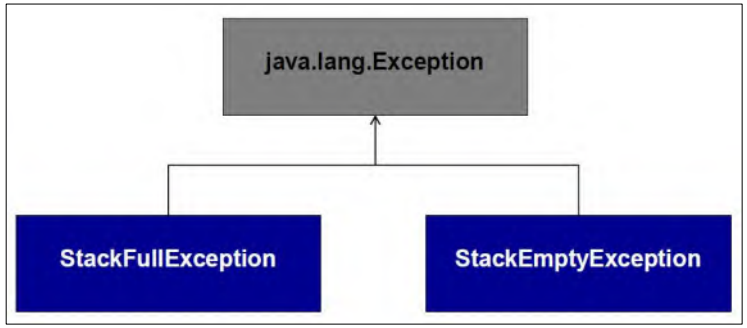
- **WO** … ist der Fehler aufgetreten
  - steht detailiert im StackTrace
  - Namen der Klassen bzw . Methoden
  - Zeilennummern vom Quellcode
    (sofern das Programm mit Debug Infos kompiliert wurde)

EXCEPTION KEYWORDS

- **WARUM** … ist der Fehler aufgetreten
  - mehr Details zur Fehlerursache
  - aus Message der Exception ablesbar
    - z.B: stack is full max size is 2

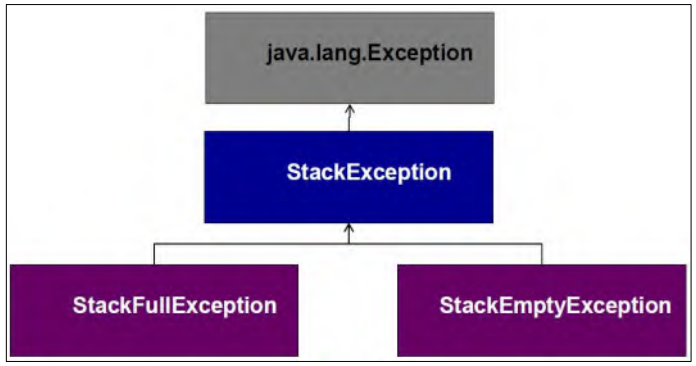
EXCEPTION HIERARCHIEN

- **Ableitung**
  - is-a Beziehung auch für Exceptions anwendbar



EXCEPTION HIERARCHIEN

- **Ableitung**
  - gemeinsame Basisklasse StackException möglich



EXCEPTION HIERARCHIEN

- **Vorteil**
  - einfache / kürzere Fehlerbehandlung möglich
  - statt konkretem Subtype ein catch von Basisklasse

```
try {
    stack.push(new Note(LocalDate.now(),
                        "TODO","DESC"));
    System.out.println(stack.pop());
    System.out.println(stack.pop());
} catch (StackException e) {
    e.printStackTrace();
}
```

EXCEPTION HIERARCHIEN

- **Nachteil**
  - falls doch andere Behandlung erforderlich müsste man innerhalb des catch Blocks manuell differenzieren *(instanceof / casting)*
  - wenn nur die Basisklasse (=StackException) verfügbar dann fehlt Detailinformation → Informationsverlust und keine sinnvolle Differenzierung möglich

EXCEPTION HIERARCHIEN

- **auch Mischformen sind möglich**

```
try {
      stack.push(new Note(LocalDate.now(),
                                  "TODO","DESC"));
      System.out.println(stack.pop());
      System.out.println(stack.pop());
} catch (StackEmptyException e) {
      e.printStackTrace();
} catch (StackException e) {
      e.printStackTrace();
}
```


EXCEPTION HIERARCHIEN

- **mehrere Catch-Blöcke: Welcher gilt/wird ausgeführt?**
  - ausgeführt wird jedenfalls nur einer
  - der 1. der basierend auf dem tatsächlichen Typ der geworfenen Exception in Frage komm
  - WICHTIG: **spezifischere Typen immer zuerst (**d.h. weiter oben) anführen, weil ein allgemeinerer Typ den spezifischeren Typ überdecken würde → Compiler Fehler

EXCEPTION HIERARCHIEN

- **Bsp. 1: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(10);
      int i = 0;
      while(i <= 10) {
            stack.push(new Note(...));
      }
      stack.pop();
} catch (StackFullException e) {
      System.err.println("stack was full");
} catch (StackEmptyException e) {
      System.err.println("stack was empty");
}
```

EXCEPTION HIERARCHIEN

- **Bsp. 1: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(10);
      int i = 0;
      while(i <= 10) {
            stack.push(new Note(...));
      }
      stack.pop();
} catch (StackFullException e) {
      System.err.println("stack was full");
} catch (StackEmptyException e) {
      System.err.println("stack was empty");
}
```

EXCEPTION HIERARCHIEN

- **Bsp. 2: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(10);
      stack.pop();
      while(true) {
            stack.push(new Note(...));
      }
} catch (StackFullException e) {
      System.err.println("stack was full");
} catch (StackEmptyException e) {
      System.err.println("stack was empty");
}
```


EXCEPTION HIERARCHIEN

- **Bsp. 2: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(10);
      stack.pop();
      while(true) {
            stack.push(new Note(...));
      }
} catch (StackFullException e) {
      System.err.println("stack was full");
} catch (StackEmptyException e) {
      System.err.println("stack was empty");
}
```

EXCEPTION HIERARCHIEN

- **Bsp. 3: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(10);
      stack.pop();
      stack.push(new Note(...));
} catch (StackEmptyException e) {
      System.err.println("stack was empty");
} catch (StackException e) {
      System.err.println("other stack failure");
}
```

EXCEPTION HIERARCHIEN

- **Bsp. 3: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(10);
      stack.pop();
      stack.push(new Note(...));
} catch (StackEmptyException e) {
      System.err.println("stack was empty");
} catch (StackException e) {
      System.err.println("other stack failure");
}
```

EXCEPTION HIERARCHIEN

- **Bsp. 4: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(2);
      stack.push(new Note(...));
      stack.push(new Note(...));
      stack.push(new Note(...));
      stack.pop();
} catch (StackFullException e) {
      System.err.println("stack was full");
} catch (StackException e) {
      System.err.println("other stack failure");
}
```


EXCEPTION HIERARCHIEN

- **Bsp. 4: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(2);
      stack.push(new Note(...));
      stack.push(new Note(...));
      stack.push(new Note(...));
      stack.pop();
} catch (StackFullException e) {
      System.err.println("stack was full");
} catch (StackException e) {
      System.err.println("other stack failure");
}
```

EXCEPTION HIERARCHIEN

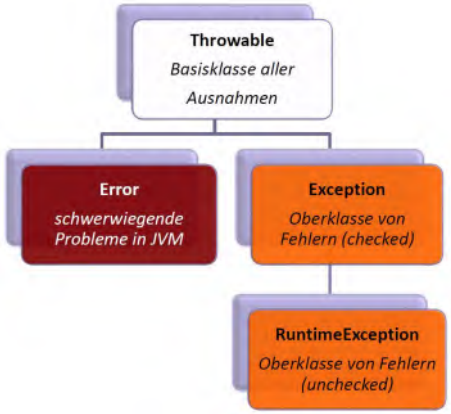
- **Bsp. 5: Welcher Catch-Block wird ausgeführt?**

```
try {
      Stack stack = new Stack(10);
      stack.pop();
} catch (StackException e) {
      System.err.println("other stack failure");
} catch (StackEmptyException e) {
      System.err.println("stack was empty");
}
```

Compile Error: Unreachable catch block for StackEmptyException
It is already handled by the catch block for StackException

EXCEPTIONS TYPHIERARCHIE

- Klassenhierarchie von Fehlern / Ausnahmen



ERRORS

- Error
  - Ableitungen von **java.lang.Error**
  - schwerwiegende Fehler die sich zumeist nicht direkt auf das Programm beziehen sondern in der Laufzeitumgebung (JVM) auftreten
  - von deratigen Fehler kann sich die Anwendung i.A. nicht „erholen“ weshalb eine Behandlung oftmals nicht sinnvoll möglich
  - ist z.B.
    - **StackOverflowError** aufgrund von Endlosrekursion
    - **OutOfMemoryError** da HeapSpace der JVM voll (RAM)

CHECKED EXCEPTIONS

- Checked Exceptions
  - Ableitung von **java.lang.Exception**
  - Ausnahmesituationen die vorherzusehen sind und sich typischerweise sinnvoll behandeln lassen
  - z.B.
    - **FileNotFoundException** weil Datei nicht existiert
    - **IOException** wenn beim Lesen/Schreiben Fehler auftritt

CHECKED EXCEPTIONS

- Catch-or-Specify Requirement
  - checked Exceptions unterliegen der Catch-or-Specify Regel d.h. Code in dem checked Exceptions auftreten können muss:
    1. entweder einen **try-catch** Block aufweisen der auf potentielle Ausnahmen reagiert / diese behandelt
    2. oder die Methode muss mittels **throws** deklarieren, welche checked Exceptions ausgelöst werden könnten die sie selbst nicht behandelt
       → COMPILER ENFORCES THESE RULES!

CHECKED EXCEPTIONS

- Checked Exceptions
  - sind Teil eines “Vertrags” (contract) zw. dem Aufrufer und der jeweiligen Methode
- Was sind die Konsequenzen?

1. beim Überschreiben von Methoden aus Supertypes
2. beim Implementieren von Methoden aus Interfaces

CHECKED EXCEPTIONS

- Konsequenzen:
  - @Override
    - eine überschriebene Methoden darf keine zusätzlichen checked Exceptions werfen
    - eine überschriebene Methoden darf aber weniger bzw. keine checked Exceptions deklarieren

  → selbiges gilt beim Implementieren von Interfaces

UNCHECKED EXCEPTIONS

- **Unchecked Exceptions**
  - Ableitung von **java.lang.RuntimeException**
  - Ausnahmesituationen im Programm welche i.A. unerwartet auftreten und sich nicht einfach behandeln lassen
  - für gewöhnlich handelt es sich um Bugs wie logische Fehler, Fehlverwendung von APIs, etc.

  → müssen NICHT(!) zwingend “gefangen” & behandelt bzw. mit throws deklariert werden

UNCHECKED EXCEPTIONS

- **Unchecked Exceptions**
  - Kennen Sie Beispiele?
  - **ArrayIndexOutOfBoundsException** wenn ein Array Zugriff basierend auf ungültigem Index erfolgt (außerhalb des Bereichs)
  - **NullPointerException** falls Program versucht mit null anstatt mit einem Objekt zu arbeiten (Methodenaufruf, Attributzugriff etc.)

EXCEPTION CHAINING

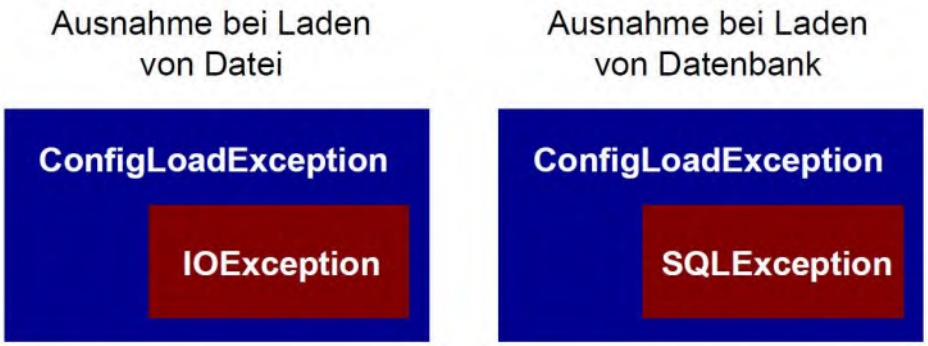
- Exception Chaining / Wrapping
  - Technik um aufgetretene Exceptions im Rahmen der Behandlung in neue (potentiell andere) Exception-Types einzupacken und erneut zu werfen
  - damit kann folglich eine Umwandlung von Exception Types erzielt werden die machmal notwendig ist
  - die ursprüngliche Exception wird im Zuge dessen als Ursache in die neue Exception eingegliedert
  - das kann mehrere Male erfolgen wodurch dann sogenanntes „Exception Chains“ daraus enstehen

EXCEPTION CHAINING

- Exception Chaining / Wrapping
  - Bsp. Laden einer Konfiguration aus Datei, Datenbank etc.
  - soll bei Fehler zu **ConfigLoadException** führen:
    - **Dateizugriff**: FileNotFoundException, IOException
    - **Datenbankzugriff**: IOException, SQLException
    - → unterschiedliche Ausnahmen werden zu **ConfigLoadException** vereinheitlicht und die ursprünglichen als Ursache (= cause) eingepackt

EXCEPTION CHAINING

- Exception Chaining / Wrapping



EXCEPTION CHAINING

- Exception Chaining / Wrapping
  - Basisklasse java.lang.Exception bietet geeignete Konstruktoren dafür an
    - **Exception (Throwable cause)**
    - **Exception (String message, Throwable cause)**

EXCEPTION CHAINING

- Exception Chaining / Wrapping

```
public class ConfigLoadException
                    extends Exception {

    public ConfigLoadException(String msg,
                        Throwable cause) {
        super(msg, cause);
    }

}
```


EXCEPTION CHAINING

- Exception Chaining / Wrapping

```
public void loadConfigFromFile() throws ConfigLoadException {
    try {
        //LOAD CONFIG FROM FILE
    } catch(IOException exc) {

        throw new ConfigLoadException("file load
                                    error", exc);
    }
}
```


EXCEPTION CHAINING

- Exception Chaining / Wrapping

```
public void loadConfigFromDB() throws ConfigLoadException {
    try {
        //LOAD CONFIG FROM DATABASE
    } catch(SQLException exc) {

        throw new ConfigLoadException("DB load
                                    error", exc);
    }
}
```


DOS & DONTS

- **Silent Catch unbedingt vermeiden**
  - leere Catch-Blöcke
  - Exceptions werden “verschluckt”
  - keine(!) Behandlung von checked Exceptions
    →macht den gesamten Mechanismus sinnlos

```
try
{
    //SOME CODE
}
catch(FileNotFoundException exc) { }
```

DOS & DONTS

- nicht **viele try-catch Blöcke** direkt hintereinander für einzelne / wenige Zeilen
  - Code wird unnötig aufgebläht → Lesbarkeit(!)
  - kein wesentlicher Informationsgewinn dadurch
  - →stattdessen:
    - ein etwas längerer try Block
    - gefolgt von ein oder mehreren catch Blöcken

DOS & DONTS

- NICHT direkt Throwable od. Exception fangen
- würde auf Errors sowie RuntimeExceptions reagieren
- dadurch werden die unterschiedlichen Fehlertypen und deren Konzepte miteinander vermischt
- **JVM Error vs. Bug vs. z.B. Fehleingabe nicht unterscheidbar**
- **Schlecht:**

```
try
{
    //SOME CODE
}
catch(Throwable t) { }
```

```
try
{
    //SOME CODE
}
catch(Exception e) { }
```


DOS & DONTS

- Stattdessen / besser:
  - auf Ausnahmen differenziert reagieren und eventuell durch re-throw / chaining vereinheitlichen

```
try {

    //SOME CODE HERE

} catch(IOException exc) {
    //handling IOException here...
} catch(ParseException exc) {
    //handling ParseException here...
}
```


DOS & DONTS

- NICHT direkt Exception werfen
  - zu wenig Information
  - nicht klar was eigentlich genau passiert ist
  - wenn dann nur aus Message (String) ersichtlich
  - würde außerdem zu “catch(Exception e)” zwingen

```
public void loadConfig() throws Exception {

    //LOADING CONFIG HERE...

}
```

DOS & DONTS

- „throw early“ Prinzip oft sinnvoll / lesbarer

```
public void printFile(String name)
                    throws IOException {

 if(name == null) {
  throw new IllegalArgumentException(“name was
                                      null”);
 }

 FileReader fr = new FileReader(name);

 //MORE CODE HERE…

}
```


VORTEILE VON EXCEPTIONS

- Vorteile des Exception Mechanismus
  - Code zur Fehlerbehandlung klar von normalem Programmcode getrennt
  - häufig keine Notwendigkeit für zahlreiche verschiedene “nichtssagende” Fehlercodes
  - Ausnahmen können entlang des Call-Stacks weitergegeben werden falls Behandlung lokal unpassend
  - sehr viel Flexibilität hinsichtlich der Gruppierung aber auch Differenzierung je nach Ausnahmen (Typen)

THREADS

Programmieren 3
Bernhard Fuchs

LERNZIELE

- Was sind Threads?
- Wie können Threads selbst implementiert werden?
- Synchronisierung von Threads

LERNHILFEN

- https://www.w3schools.com/java/java_threads.asp
- https://www.javatpoint.com/multithreading-in-java

ÜBERBLICK

- Seit etwa 1980 gibt es Multitasking-OS
- Mehrere Programme / Prozesse laufen gleichzeitig
- Ein Scheduler weißt Programmen Rechenzeit zu
- Ab etwa 2005 setzen sich auch MulticoreCPUs durch

BEISPIELE

- Bis iPhone 4, noch Einkern CPU
- Ab iPhone 4s, Mehrkern CPU
  - Anfangs 2 Kerne
  - Nun 6 Kerne
- Aktuelle Notebook CPUs
  - Bis zu 8 Kerne und mehr

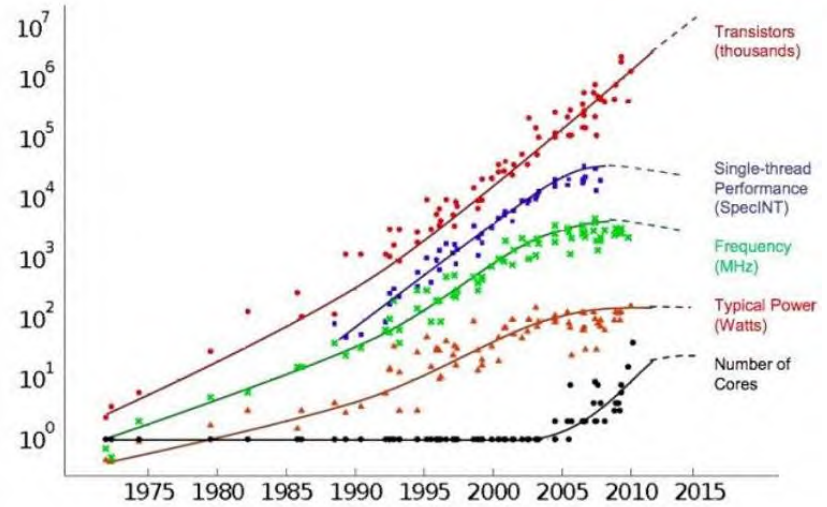
MOORESCHE GESETZ

- Was ist das?

MOORESCHE GESETZ

- Zitat Wikipedia:
  - „Das **mooresche Gesetz** (englisch Moore's law; deutsch „**Gesetz**“ im Sinne von „Gesetzmäßigkeit“) besagt, dass sich die Komplexität integrierter Schaltkreise mit minimalen Komponentenkosten regelmäßig verdoppelt; je nach Quelle werden 12 bis 24 Monate als Zeitraum genannt.

MOORESCHE GESETZ



- Quelle: https://www.skillbyte.de/kuenstliche-intelligenz-im-gewerbe-die-zukunftsaussichten/

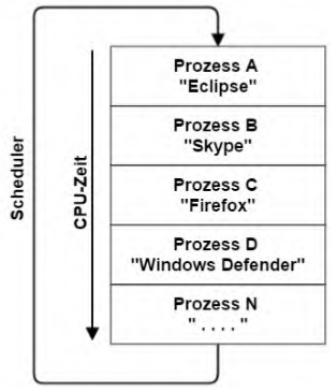
WAS IST EIN PROZESS (1/2)

- Entspricht einer Anwendung
- Hat zumindest einen Thread (Hauptthread)
- Bekommt vom OS einen Namespace und Speicher (Heap) zugeteilt
- Bekommt vom Scheduler eine gewisse Rechenzeit zugeteilt

WAS IST EIN PROZESS (2/2)

- Ein Prozess kann mit einer Anwendung verglichen werden
- Jeder Prozess bekommt abwechselnd CPU-Zeiten zugeteilt
- Prozesse sind voneinander unabhängig

WAS IST EIN PROZESS



WAS IST EIN THREAD

- Ist im Prozess „**gefangen**“
- Teilt sich mit anderen Threads dem Prozess zur Verfügung gestellten Speicher
- Nutzen die globalen Daten eines Prozesses
- Sind „leichtgewichter“ und einfacher zu erzeugen
- Thread-Wechsel **NICHT vorhersagbar**

EINSATZGEBIETE VON THREADS

- Hintergrundprozesse
- Usability
- Performancevorteile durch Ausnützung von CPUs / Kernen
- „Anforderung“ an die Software / Lösung

SYNONYME

- Faden
- Ausführungsstrang
- Nebenläufigkeit

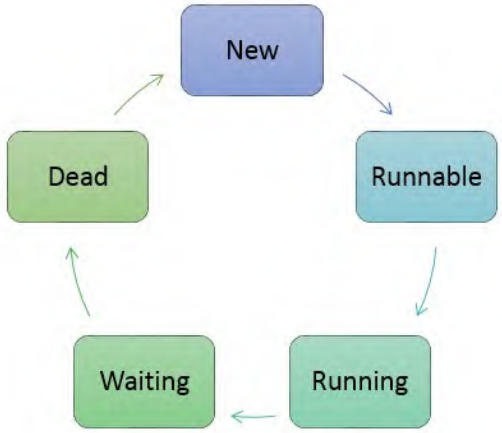
WARUM THREADS (1/2)

- Java unterstützt die **nebenläufige** Programmierung direkt
- Das Konzept beruht auf so genannten Threads, das sind **parallel ablaufende** Aktivitäten, die sehr schnell umschalten

WARUM THREADS (2/2)

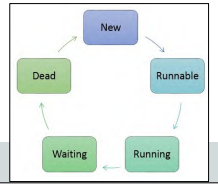
- Threads werden entweder direkt vom **Betriebssystem** unterstützt oder von der **virtuellen Maschine simuliert**
- Sehr einfach zu verwenden

THREAD LIFE CYCLE



THREAD LIFE CYCLE

- **NEW**: In dieser Phase wird der Thread mit der Klasse "Thread-Klasse" erstellt. Er bleibt in diesem Zustand, bis das Programm den Thread startet. Es ist auch als geborener Faden bekannt.
- **RUNNABLE**: Auf dieser Seite wird die Instanz des Threads mit einer Startmethode aufgerufen. Die Thread-Steuerung wird dem Scheduler übergeben, um die Ausführung zu beenden. Es hängt vom Scheduler ab, ob der Thread ausgeführt werden soll.
- **RUNNING**: Wenn der Thread ausgeführt wird, wird der Status in den Status "Ausführen" geändert. Der Scheduler wählt einen Thread aus dem Thread-Pool aus und beginnt mit der Ausführung in der Anwendung.
- **WAITING**: Dies ist der Zustand, in dem ein Thread warten muss. Da in der Anwendung mehrere Threads ausgeführt werden, ist eine Synchronisierung zwischen den Threads erforderlich. Daher muss ein Thread warten, bis der andere Thread ausgeführt wird. Daher wird dieser Zustand als Wartezustand bezeichnet.
- **DEAD**: Dies ist der Zustand, in dem der Thread beendet wird. Der Thread befindet sich im laufenden Zustand und befindet sich nach Abschluss der Verarbeitung im "toten Zustand".



THREADS – LIVE-DEMO

- Live-Demo Beispiel wird im Nachhinein auf Moodle zu finden sein.
  - Demo: SingleThread
  - Weitere Beispiele in Package: threads

SINGLE THREAD

- Ein einzelner Thread ist im Grunde eine leichte und kleinste Verarbeitungseinheit.
- Java verwendet Threads mithilfe einer "Thread-Klasse".

SINGLE THREAD - VORTEILE

- Vorteile eines einzelnen Threads:
  - Reduziert den Overhead in der Anwendung, wenn einzelne Threads im System ausgeführt werden
  - Außerdem werden die Wartungskosten der Anwendung reduziert.

THREADS ERSTELLEN

- 2 Möglichkeiten:
  - Java Thread by extending Thread Class (**extends Thread**)
  - Java Thread by implementing Runnable interface (**implements Runnable**)

KLASSE THREAD

- Ableiten von der Klasse Thread
- In Thread auszuführender Code in Methode **run()** implementieren oder in der Methode run() aufrufen

INTERFACE RUNNABLE

- Alternative zur Klasse Thread (z.B.: Mehrfachvererbung)
- Run() muss implementiert werden
- Die Klasse Thread dient als Hilfsklasse für Start und Stopp

RUNNABLE VS. THREAD

- Die Implementierung von Runnable ist die bevorzugte Methode. Hier spezialisieren oder ändern Sie das Verhalten des Threads nicht wirklich. Sie geben dem Thread nur etwas zum Ausführen. Das heißt, Komposition ist der bessere Weg.
- Java unterstützt nur die Einzelvererbung, sodass Sie nur eine Klasse erweitern können.
- Durch das Implementieren einer Schnittstelle wird die Trennung zwischen Ihrem Code und der Implementierung von Threads sauberer.
- Durch die Implementierung von Runnable wird Ihre Klasse flexibler. Wenn Sie Thread erweitern, befindet sich die Aktion, die Sie ausführen, immer in einem Thread. Wenn Sie Runnable implementieren, muss dies jedoch nicht der Fall sein.

MULTITHREADS

- MULTITHREADING in Java ist ein Prozess, bei dem zwei oder mehr Threads gleichzeitig ausgeführt werden, um die CPU optimal zu nutzen.
- Multithread-Anwendungen führen zwei oder mehr Threads aus, die gleichzeitig ausgeführt werden.
- Daher wird es in Java auch als Parallelität bezeichnet. Jeder Thread läuft parallel zueinander.
- Mehrere Threads weisen keinen separaten Speicherbereich zu, daher sparen sie Speicher.
- Außerdem dauert das Wechseln des Kontexts zwischen Threads weniger Zeit.

MULTITHREADS - VORTEILE

- Vorteile eines von Multi Threads:
  - Die Benutzer werden nicht blockiert, da Threads unabhängig sind und wir gleichzeitig mehrere Vorgänge ausführen können
  - Da die Threads unabhängig sind, werden die anderen Threads nicht betroffen, wenn ein Thread eine Ausnahme erfüllt.

THREADS – LIVE-DEMO

- Live-Demo Beispiel wird im Nachhinein auf Moodle zu finden sein.
  - Demo:
    - RunnableExample
    - ThreadExample
    - Package: example1 (demo_1-4_von_folien.zip)
  - Weitere Beispiele in Package: threads

RUNNABLE CODE EXAMPLE

```
public class Person implements Runnable {

    private String name;

    public Person(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(name);
            try {
                Thread.sleep( millis: 100);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```


RUNNABLE CODE EXAMPLE

```
public class main {
    public static void main(String[] args) {
        Runnable person1 = new Person( name: "Name1");
        Runnable person2 = new Person( name: "Name2");

        Thread thread1 = new Thread(person1);
        Thread thread2 = new Thread(person2);

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

THREAD SCHEDULER IN JAVA

- Der Thread-Scheduler in Java ist der Teil der JVM, der entscheidet, welcher Thread ausgeführt werden soll.
- Es gibt keine Garantie dafür, welcher ausführbare Thread vom Thread-Scheduler ausgeführt wird.
- Es kann jeweils nur ein Thread in einem Prozess ausgeführt werden.

WICHTIGE METHODEN

- **.start()** – startet den Thread
  - -> Methode **run()** wird vom Scheduler aufgerufen
- **.join()** - wartet bis der Thread fertig ist
- **.sleep()** – legt die Arbeit innerhalb eines Threads für eine gewisse Zeit nieder
- **.yield()** - legt die Arbeit innerhalb eines Threads nieder und wartet bis zum nächsten Zeitschlitz
- **.getName()** – retourniert den Namen eines Threads
- **.setPriority()** – ändert die Priorität eines Threads
- **.wait()** - der aktuelle Thread wird gezwungen zu warten, bis ein anderer Thread notify () oder notifyAll () für dasselbe Objekt aufruft.
- **.notify()** - Die notify () -Methode wird zum Aufwecken von Threads verwendet, die auf einen Zugriff auf den Monitor dieses Objekts warten.

SLEEP METHODE

- Die sleep () -Methode der Thread-Klasse wird verwendet, um einen Thread für die angegebene Zeitspanne in den Ruhezustand zu versetzen.
- Syntax der sleep () -Methode in Java
  - Die Thread-Klasse bietet zwei Methoden zum Schlafen eines Threads:
    - public static void sleep(long miliseconds)throws InterruptedException
    - public static void sleep(long miliseconds, int nanos)throws InterruptedException

JOIN() METHODE

- Die join () -Methode wartet darauf, dass ein Thread stirbt.
- Mit anderen Worten, die aktuell ausgeführten Threads werden nicht mehr ausgeführt, bis der Thread, mit dem sie verbunden sind, ihre Aufgabe abgeschlossen hat.

THREAD ZWEIMAL STARTEN?

- Nein. Nach dem Starten eines Threads kann dieser nie wieder gestartet werden.
  - In diesem Fall wird eine **IllegalThreadStateException** ausgelöst.
- In diesem Fall wird der Thread einmal ausgeführt, aber zum zweiten Mal wird eine Ausnahme ausgelöst.

PRIORITÄT EINES THREADS

- Jeder Thread hat eine Priorität. Prioritäten werden durch eine Zahl zwischen 1 und 10 dargestellt.
- In den meisten Fällen plant der Thread-Zeitplan die Threads entsprechend ihrer Priorität (als präemptive Planung bezeichnet).
- Es kann jedoch nicht garantiert werden, da es von der JVM-Spezifikation abhängt, welche Zeitplanung gewählt wird.
- 3 Konstanten sind in der Thread-Klasse definiert:
  - public static int MIN_PRIORITY
  - public static int NORM_PRIORITY
  - public static int MAX_PRIORITY
  - Die Standardpriorität eines Threads ist 5 (NORM_PRIORITY). Der Wert von MIN_PRIORITY ist 1 und der Wert von MAX_PRIORITY ist 10.

HÖFLICHES STOPPEN

- Wird über eigene Methode sichergestellt
- Innerhalb von Person Klasse wird auf Status oder sonstiges Ereignis abgefragt

AUFGABE

- Erstellen Sie folgendes Programm:
  - Main: einen zweiten Thread starten der 10 Sekunden lange benötigt bis er sich beendet.
  - Beide Threads sollen beim Starten ihren Namen ausgeben.
  - benutzen Sie das Runnable Interface

THREADS – LIVE-DEMO

- Live-Demo Beispiel wird im Nachhinein auf Moodle zu finden sein.
  - Muss nicht mitgeschrieben werden.
    - Demo:
      - Package: example2 (Höfliches Stoppen)
    - Weitere Beispiele in Package: threads

HÖFLICHES STOPPEN

```
private boolean isRunning = true;

public void requestShutDown() {
    isRunning = false;
}

@Override
public void run() {
    while (isRunning) {...}
}
```


UNHÖFLICHES STOPPEN

- Thread.stop();
- Beendet den Thread gewaltsam
- Ist deprecated/veraltet/überholt
- Keine wirkliche saubere Alternative (außer sauber zu programmieren)

KOMBINATION

- Person1.requestShutDown();
- thread1.join(5000); // 5 sec.
- Wenn immer noch nicht fertig:
  - thread1.stop();

ÜBUNG UE-3-1

- Schreiben Sie eine Konsolenanwendung mit einem Hintergrund-Thread, welcher im Sekundentakt die aktuelle Uhrzeit ausgibt
- Aktuelle Uhrzeit:
  - Date d = new Date();
  - System.out.println(d.toString())
- Per Tastendruck soll die Anwendung (und der Thread „höflich“) gestoppt werden können.
- Lösung: später auf Moodle

KRITISCHE SEKTIONEN

- Codeteile, **die nicht gleichzeitig von mehreren Threads** ausgeführt werden dürfen, müssen gesperrt werden!
- Weil sich die Threads „ansonsten in die Quere kommen würden“.

THREADS – LIVE-DEMO

- Live-Demo Beispiel wird im Nachhinein auf Moodle zu finden sein.
  - Demo:
    - Package: example3 (Kritische Sektionen)
  - Weitere Beispiele in Package: threads

SYNCHRONIZED

- synchronized-Block darf nur von einem Thread zu einer Zeit betreten werden
- Andere Threads warten
- Jedes Objekt kann ein Sperr-Objekt sein
- Synchronized kann **innerhalb von Methoden** (Variante A) oder
- auch auf **gesamte Methoden (Variante B)** angewendet werden

THREADS – LIVE-DEMO

- Live-Demo Beispiel wird im Nachhinein auf Moodle zu finden sein.
  - Demo:
    - Package: example4 (Synchronized innerhalb Methoden)
  - Weitere Beispiele in Package: threads

SYNCHRONIZED

```
public void run() {

    for (int i = 0; i < 1_000_000; i++) {
        // Variante A - Kritischer CodeTeil
        synchronized (LockObject) {
            Increase();
        }
    }

}

// Variante B - Direkt bei kritischer Methode
private synchronized void Increase() {
    Counter++;
}
```


DEADLOCKS

- Tritt immer auf, wenn sich Lock-Objekte in die Quere kommen
- Thread **A hat Lock auf X** und möchte Lock auf **Y** haben
- Thread **B hat bereits Lock auf Y** und möchte Lock auf **X** haben

DEADLOCKS, MÖGLICHE LÖSUNGEN:

- Locks immer in derselben Reihenfolge anfordern
- Alle Locks am Beginn gleichzeitig Anforderung
- Großzügiger sperren (Nachteil für Performance)
- Deadlock erkennen und Prozess neu starten
- Locks verhindern:
  - Daten klonen und doppelt ausführen, später zusammenführen

ÜBUNG UE-3-**2**

- Erweitern Sie Ihr „Uhr-Beispiel“, sodass neben der Uhrzeit auch die Anzahl an CPUs und der noch aus Sicht von Java freie Speicher ausgegeben wird:

```
Date d = new Date();
System.out.print("[");
System.out.print(d.toString());
System.out.print(", CPUS: ");
System.out.print(Runtime.getRuntime().availableProcessors());
System.out.print(", FreeMem: ");
System.out.print(Runtime.getRuntime().freeMemory());
System.out.print("]");
```

- Starten Sie zwei „Uhr-Threads“ gleichzeitig
- Entdecken Sie, dass sich die beiden Threads in die Quere kommen?
- Definieren Sie die Ausgabe als „Kritische Sektion“ mit synchronized
- Führen Sie die Anwendung erneut aus und stellen sie fest, dass sich die Threads nicht mehr in die Quere kommen.
- Lösung: threads/uebungen/ue2

ÜBUNG UE-3-**3**

- Schreiben Sie eine einfache Klasse, welche Runnable implementiert und innerhalb der run()-Methode einen statischen Counter der Klasse erhöht.
  - Pro Sekunde soll einmal der Counter erhöht werden
  - Bis die Zahl 20 erreicht ist
- Erstellen Sie zwei Thread-Instanzen auf einem Worker
- Behandeln Sie den Counter als „Kritische Sektion“ mit synchronized
- Stellen Sie sicher, dass am Ende das richtige Ergebnis herauskommt (Counter muss 20 sein)
  - Geben Sie parallel auf der Console den Threadnamen sowie den aktuellen Counter Wert aus. (In der Run Methode)
    - Was können Sie hier feststellen?
- Lösung im Moodle

ÜBUNG UE-3-**4**

- Erweitern Sie das vorherige Beispiel so, dass abwechselnd Threads eingesetzt werden.
- Tipp: Verwenden Sie die Methoden: wait() sowie notify()
- Tipp: Geben Sie übe die Main Klasse den Wert der Variable "counter" auf der Konsole aus. - Um sicher zu gehen, dass diese 20 erreicht hat.
- Lösung im Moodle

NETWORK IN JAVA

Programmieren 3
Bernhard Fuchs

LERNZIELE

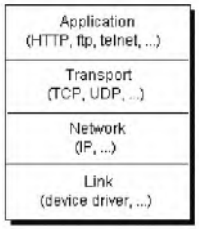
- Die Verwendung der Klasse URL kennen
- Netzwerkkommunikation mittels Sockets und Java IO verwenden können
- Serverapplikationen mit Hilfe von Server Sockets implementieren können
- Verteilte Applikationen mit parallelisierter Verarbeitung erstellen können

LERNHILFEN

- https://www.javatpoint.com/java-networking

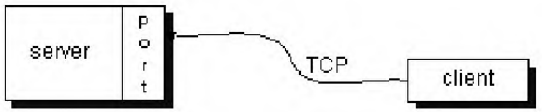
NETZWERK ALLGEMEIN

- Java Programme befinden sich im Application Layer
- Abhängig vom verwendeten Protokoll (TCP oder UDP) werden unterschiedliche Klassen verwendet



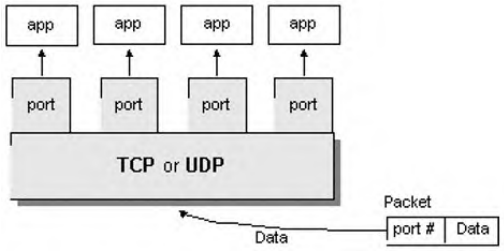
PORTS

- Auf einem Rechner, bzw. einer IP-Adresse, werden unterschiedliche Ports verwendet, um die Zuordnung von Datenpaketen zu Applikationen zu steuern
- Eine Serverapplikation hört auf einem Port auf eingehende Verbindungen eines Clients



PORTS

- 65535 Ports (16 bit) sind verfügbar
- Die ersten 1024 sind reserviert (HTTP, FTP, …)



PORTS

- HTTP – 80
- HTTPS – 443
- SSH – 22
- FTP - 21

PACKAGE JAVA.NET

- Die wichtigsten Klassen in diesem Package:
  - `URL`
  - `URLConnection`
  - `Socket`
  - `ServerSocket`
  - `DatagramPacket`
  - `DatagramSocket`
  - `MulticastSocket`

TCP

UDP

URL

URL

- **U**niform **R**esource **L**ocator
- Repräsentieren Ressourcen im Netzwerk (z.B. Webseiten oder Datenbankabfragen)
- Zwei wesentliche Komponenten:
  - Protocol identifier: http
  - Resource name – www.wetter.at
    - Host Name: www.wetter.at
    - Filename /wetter/…/graz/index.html
    - Port Number (optional): 80
    - Reference (optional) # Prognose

URL - KONSTRUKTOREN

- **URL myURL =**
  - **new URL ("http://www.wetter.at/…/index.html")**
  - **new URL („http", „www.wetter.com", „/wetter/oesterreich/steiermark/graz/index.html")**

URL – LESEN VON URL

- **InputStream openStream() throws IOException**
- Liefert einen InputStream zum Auslesen des Dokuments. Dasselbe geht mit: **myURL.openConnection().getInputStream();**
- Die weitere Verarbeitung funktioniert analog wie das Auslesen einer Datei:
- **BufferedReader br = new BufferedReader(new InputStreamReader(myURL.openStream()));**

URL – LIVE-DEMO

- Live-Demo Beispiel wird im nachhinein auf Moodle zu finden sein
  - Demo:
    - Network → beispiel1/ReadFromUrl.java
  - Weitere Beispiele in Package: network

URL – UE 1

- Erstellen Sie ein Programm, das eine Internetadresse aus einer Textdatei liest, und den Inhalt dieser URL in eine Datei “content.html” schreibt.
- Lösung: network/beispiel1/ReadFromUrlAndWrite.java

URLConnection

URLCONNECTION

- **URLConnection myConn =**
  - **myURL.openConnection();**
  - **myConn.setDoOutput (true); //um schreiben zu können**
  - **myConn.getOutputStream ();**
  - **myConn.getInputStream ();**

UE-2

- Analysieren Sie das Beispiel im package *network/beispiel2*
- Kommentieren Sie die Zeile
  - “conn.setDoOutput(true);”
- aus und führen Sie das Programm aus. Was passiert?

Sockets

SOCKETS

- Sind die Endpunkte einer TCP Verbindung
- Sockets werden an Ports gebunden
- Ein Server hört an einem bekannten Port auf eingehende Verbindungen
- Der Client öffnet ein Socket und baut eine Verbindung zum Server auf

SOCKETS

- Der Server akzeptiert die Verbindung
- Am Server wird ein neues Socket für die bestätigte Verbindung erzeugt
- Am bekannten Port wird weiterhin auf eingehende Verbindungen gewartet

SOCKETS

- **Socket mySocket =**
  - **new Socket(); //unconnected**
  - **new Socket ("www.wetter.at", 80);**
  - **byte[] remAdr = {173,194,35,152}; new Socket (InetAdress.getByAdress(remAdr), 80);**
  - **byte[] locAdr = {10,124,79,0}; new Socket (InetAdress.getByAdress(remAdr), 80; InetAdress.getByAdress(locAdr), 8000);**

SOCKETS – LESEN

- **InputStream getInputStream() throws IOException**
- Liefert einen InputStream zum Auslesen des Dokuments. Wird der InputStream geschlossen, so wird auch der Socket geschlossen.
- Die weitere Verarbeitung funktioniert analog wie das Auslesen einer Datei:
  - **BufferedReader br = new BufferedReader(new InputStreamReader(mySocket.getInputStream()))**

SOCKETS – SCHREIBEN

- **OutputStream getOutputStream() throws IOException**
  Liefert einen InputStream zum Auslesen des Dokuments. Wird der OutputStream geschlossen, so wird auch der Socket geschlossen.
- Die weitere Verarbeitung funktioniert analog wie das Schreiben in eine Datei:
  - **BufferedWriter br = new BufferedWriter(new OutputStreamWriter(mySocket.getOutputStream()));**

SOCKETS – UE 3

- Analysieren Sie das Beispiel im package network/beispiel3
- Schreiben Sie ein Programm, das sich mit der Internetadresse time-a.timefreq.bldrdoc.gov auf Port 13 oder 37 verbindet und dann die aktuelle Zeit liest

ServerSocket

SERVERSOCKET

- **ServerSocket mySocket =**
  - **new ServerSocket(); //unbound**
  - **new ServerSocket (9090)**

SERVERSOCKET

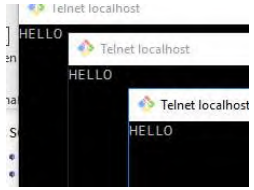
- Akzeptieren eingehender Verbindungen
- **Socket accept() throws IOException**
- Wartet auf eingehende Verbindungsanfragen und stellt die Verbindung her. Der serverseitige Endpunkt der Verbindung ist eine neue Socket Instanz (diese wird zurückgegeben).

SERVERSOCKET – UE-4

- Analysieren Sie das Beispiel im package network/beispiel4.
- Adaptieren Sie das Server-Programm so, dass mehr als eine Client-Verbindung aufgenommen werden kann. Testen Sie Ihre Implementierung mit Hilfe von telnet
  - https://windowsreport.com/telnet-windows-10/
  - Via CommandLine (cmd): `telnet localhost 9090`
- Lösung: network/beispiel4/loesung

SERVERSOCKET – LÖSUNG 4

- Erweitern Sie Ihren Server so, dass mehrere Clientverbindungen parallel behandelt werden.
  - Test mittels mehreren Telnet Instanzen (telnet localhost 9090)



- Lösung: network/beispiel4/loesung