

Lektion 17

Exceptions II
Streams

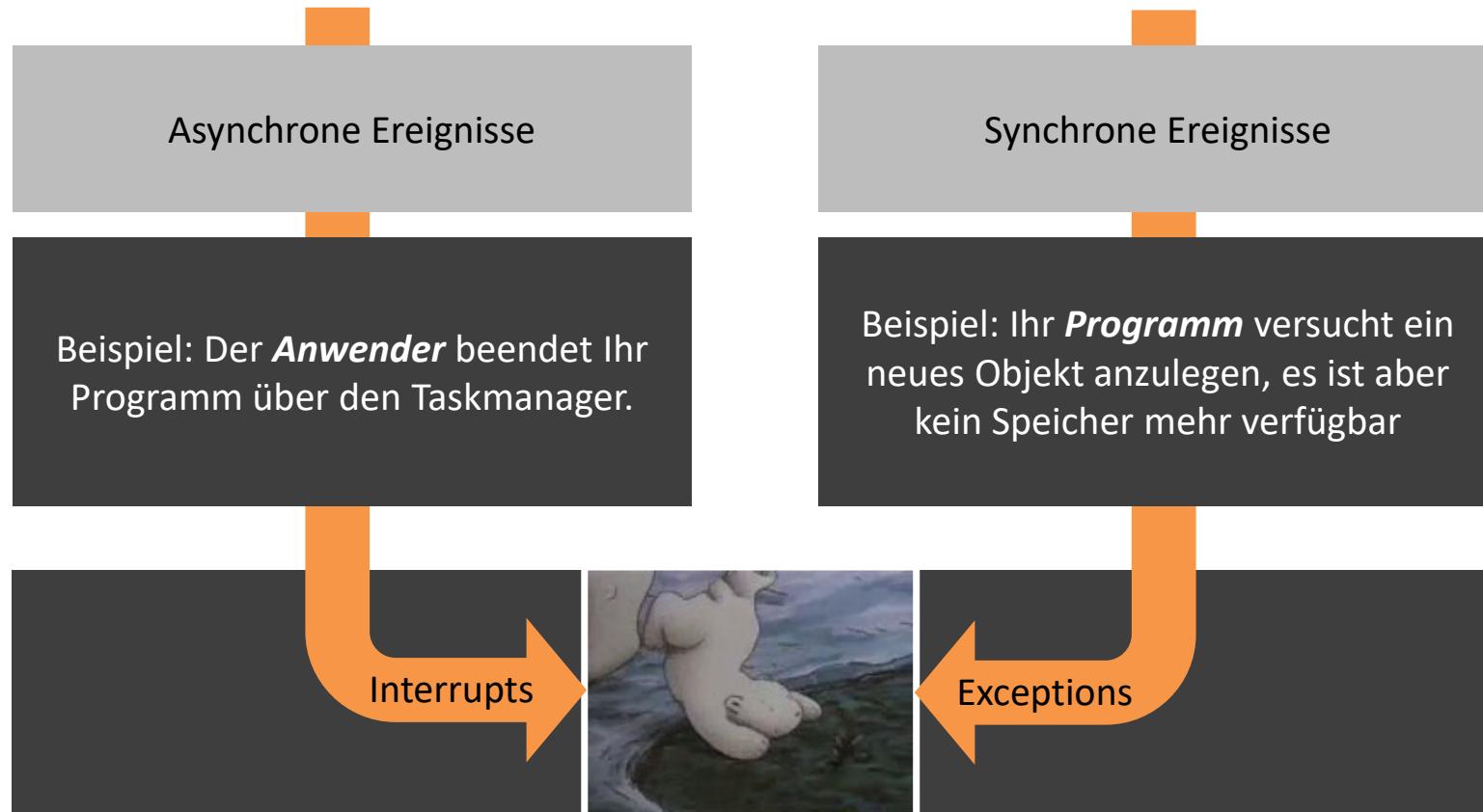
Exceptions II

Sequenzieller Programmablauf

Normalerweise führt die Java Laufzeitumgebung Ihr Programm Schritt für Schritt aus.
Doch was passiert, wenn etwas Unvorhergesehenes geschieht?



Was kann den sequenziellen Programmablauf stören?



Wie kann mit Interrupts und Exceptions umgegangen werden?

Asynchrone Ereignisse
(Interrupts)



- Als Programmierer hat man kaum eine Chance, unmittelbar zu reagieren.
- Reaktion auf Interrupts sind betriebssystemspezifisch und mit Java nicht ohne Weiteres möglich.
 - Wichtige Programmdaten sollten häufig zwischengespeichert werden.

Synchrone Ereignisse
(Exceptions bzw.
Ausnahmen)



- „Verdächtige“ Stellen im Programm, an denen Exceptions auftreten können, sind prinzipiell bekannt.
- Hierauf kann reagiert werden.

```
public static void main(String[] args)
{
    Scanner scanner = new Scanner(System.in);
    System.out.println("Bitte geben Sie eine Zahl ein: ");
    int zahl = scanner.nextInt();
    System.out.println(zahl);
    scanner.close();
}
```

Was passiert bei der Eingabe
von Buchstaben?

Bitte geben Sie eine Zahl ein:

ew

```
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:864)
at java.util.Scanner.next(Scanner.java:1485)
at java.util.Scanner.nextInt(Scanner.java:2117)
at java.util.Scanner.nextInt(Scanner.java:2076)
at exceptions.ScannerExceptionTest.main(ScannerExceptionTest.java:11)
```



Wir haben eine Exception ausgelöst!

Was passiert beim Auslösen einer Exception?



Das Laufzeitsystem unterbricht den normalen Ablauf.



Es generiert ein Exception-Objekt, in dem es genau festhält, wann, wo und was schief gegangen ist.



Dieses Exception-Objekt wird „geworfen“ in der Hoffnung, dass es jemand auffängt und eine Lösung parat hat.



Falls es niemand fängt, beendet das Laufzeitsystem das Programm.

Das Auslösen einer Exception nennt man auch „Werfen“ einer Exception.

Wie können wir eine geworfene Exception fangen?

Mit einem try-catch Block bestehend aus

- dem Schlüsselwort **try** gefolgt von einem Block
- dem Schlüsselwort **catch** gefolgt vom Exceptionnamen, gefolgt von einem Block

```
try
{
    //Anweisungen, die wir versuchen wollen, auszuführen.
}
catch(<class name of exception> e)
{
    //Anweisungen, die wir beim Auftreten eines Fehlers im try-Block ausführen
}
```

Wie können wir eine geworfene Exception fangen?

```
Scanner scanner = new Scanner(System.in);
System.out.println("Bitte geben Sie eine Zahl ein: ");
try
{
    int zahl = scanner.nextInt();
    System.out.println(zahl);
}
catch(InputMismatchException e)
{
    System.out.println("Fehler beim Einlesen der Zahl!");
}
scanner.close();
```

Code, den wir versuchen auszuführen

Fehlerfall, den wir behandeln wollen

Code, den wir im Fehlerfall ausführen

Wie können wir den Fehler beheben?

```
Scanner scanner = new Scanner(System.in);
boolean fehlerGefunden;
do
{
    try
    {
        System.out.println("Bitte geben Sie eine Zahl ein: ");
        int zahl = scanner.nextInt();
        fehlerGefunden = false;
        System.out.println(zahl);
    }
    catch (InputMismatchException e)
    {
        System.out.println("Fehler beim Einlesen der Zahl!");
        scanner.nextLine();
        fehlerGefunden = true;
    }
}
while (fehlerGefunden);
scanner.close();
```

kein Fehler aufgetreten

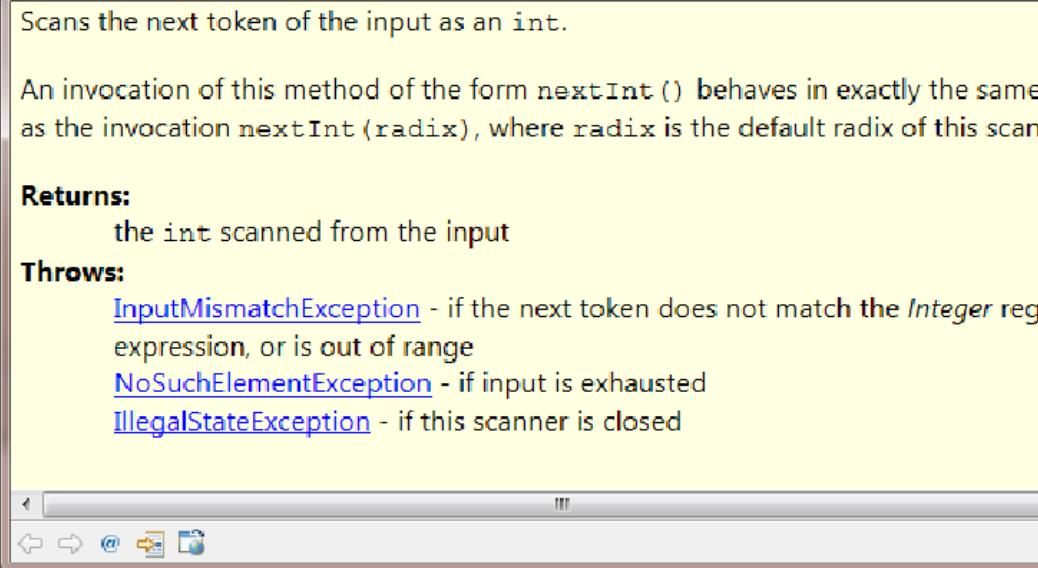
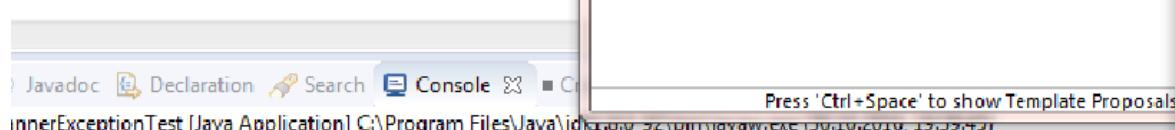
lies den fehlerhaften Input, ansonsten
wirft nextInt() den gleichen Fehler, weil
der Eingabepuffer unverändert ist.

wiederhole solange ein Fehler auftritt

Woher wissen wir überhaupt, wo Fehler auftreten können?

Java Doc

```
Scanner scanner = new Scanner(System.in);
System.out.println("Bitte geben Sie eine Zahl ein: ");
try
{
    int zahl = scanner.nextInt();
    System.out.println(zahl);
}
catch(InputMismatchException)
{
    System.out.println("Fehler");
}
scanner.close();
```



Um uns das “Weglesen” der fehlerhaften Eingabe zu sparen, kann man auch direkt alle Eingaben als String lesen und in einen Integer umwandeln:

```
Scanner scanner = new Scanner(System.in);
System.out.println("Bitte geben Sie eine Zahl ein: ");
String eingabe = scanner.nextLine();
int zahl = Integer.valueOf(eingabe);
scanner.close();
```

Wenn in `eingabe` Nichtzahlen auftreten, löst `Integer.valueOf` eine Exception aus:

Bitte geben Sie eine Zahl ein:

afs

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "afs"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.valueOf(Integer.java:766)
at exceptions.ScannerExceptionTest.testWithNextLineOnly(ScannerExceptionTest.java:75)
at exceptions.ScannerExceptionTest.main(ScannerExceptionTest.java:13)
```

```
Scanner scanner = new Scanner(System.in);
boolean fehlerGefunden;
do
{
    try
    {
        System.out.println("Bitte geben Sie eine Zahl ein: ");
        String eingabe = scanner.nextLine();
        int zahl = Integer.valueOf(eingabe);
        fehlerGefunden = false;
        System.out.println(zahl);
    }
    catch (NumberFormatException e)
    {
        System.out.println("Fehler beim Einlesen der Zahl!");
        fehlerGefunden = true;
    }
}
while (fehlerGefunden);
scanner.close();
```

Wo können Exceptions auftreten?

**In von uns aufgerufenen
fremden Methoden**

**In unserem eigenen
Programmcode**

Wann?

Bei Programmierfehlern

Wenn beim Methodenaufruf die notwendigen Vorbedingungen nicht hergestellt wurden.

Wenn Hardware ausfällt oder Ressourcen knapp werden.

Wir können die Exception fangen, um den Abbruch zu verhindern.

Wir können die Exception werfen, um das Problem mitzuteilen.

```
public static int tageImMonat(String monat)
{
    int tage = switch(monat)
    {
        case "Februar" -> 28;
        case "April", "Juni", "September", "November" -> 30;
        case "Januar", "März", "Mai", "Juli", "August",
            "Oktober", "Dezember" -> 31;
        default -> -1;
    };
    return tage;
}
```

Was passiert, wenn ein falscher Monat übergeben wird?

Die Methode sollte besser einen Fehler auslösen.

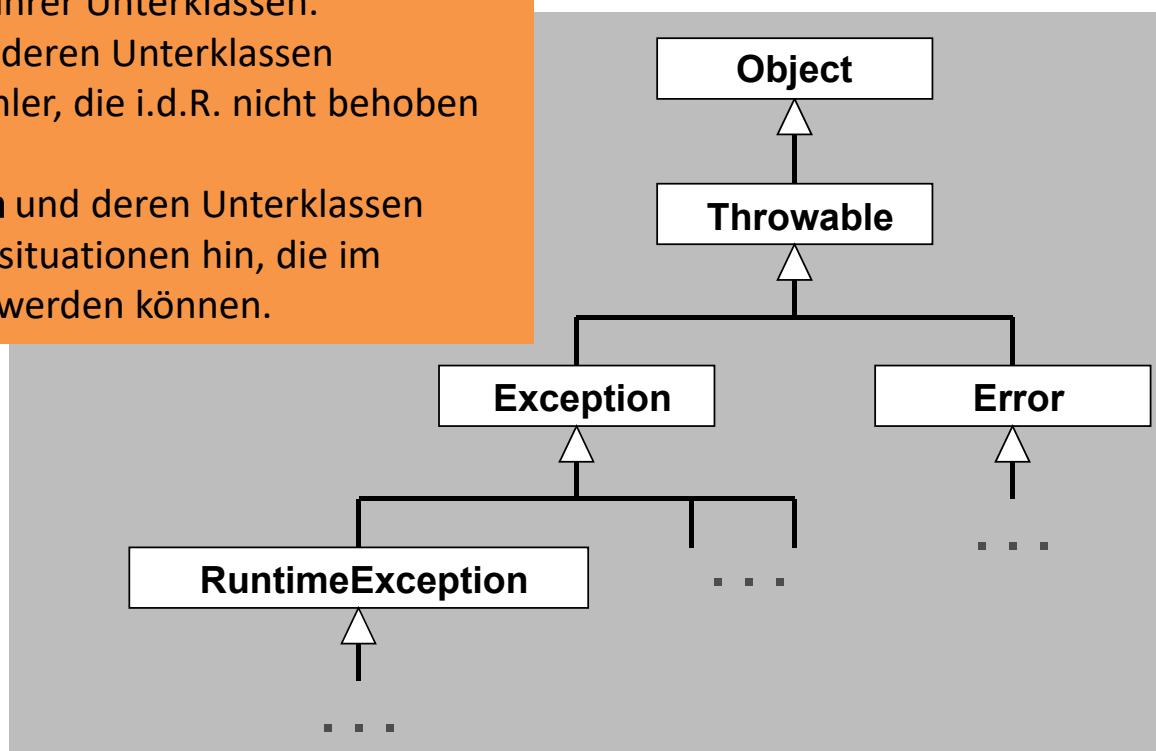
```
public static int tageImMonat(String monat) throws Exception
{
    int tage = switch(monat)
    {
        case "Februar" -> 28;
        case "April", "Juni", "September", "November" -> 30;
        case "Januar", "März", "Mai", "Juli", "August",
            "Oktober", "Dezember" -> 31;
        default -> throw new Exception("falscher Monatsname");
    };
    return tage;
}
```

Ankündigung, dass in dieser Methode eine Exception auftreten kann, erfolgt über das **throws**-Schlüsselwort, gefolgt vom Klassennamen der Exception.

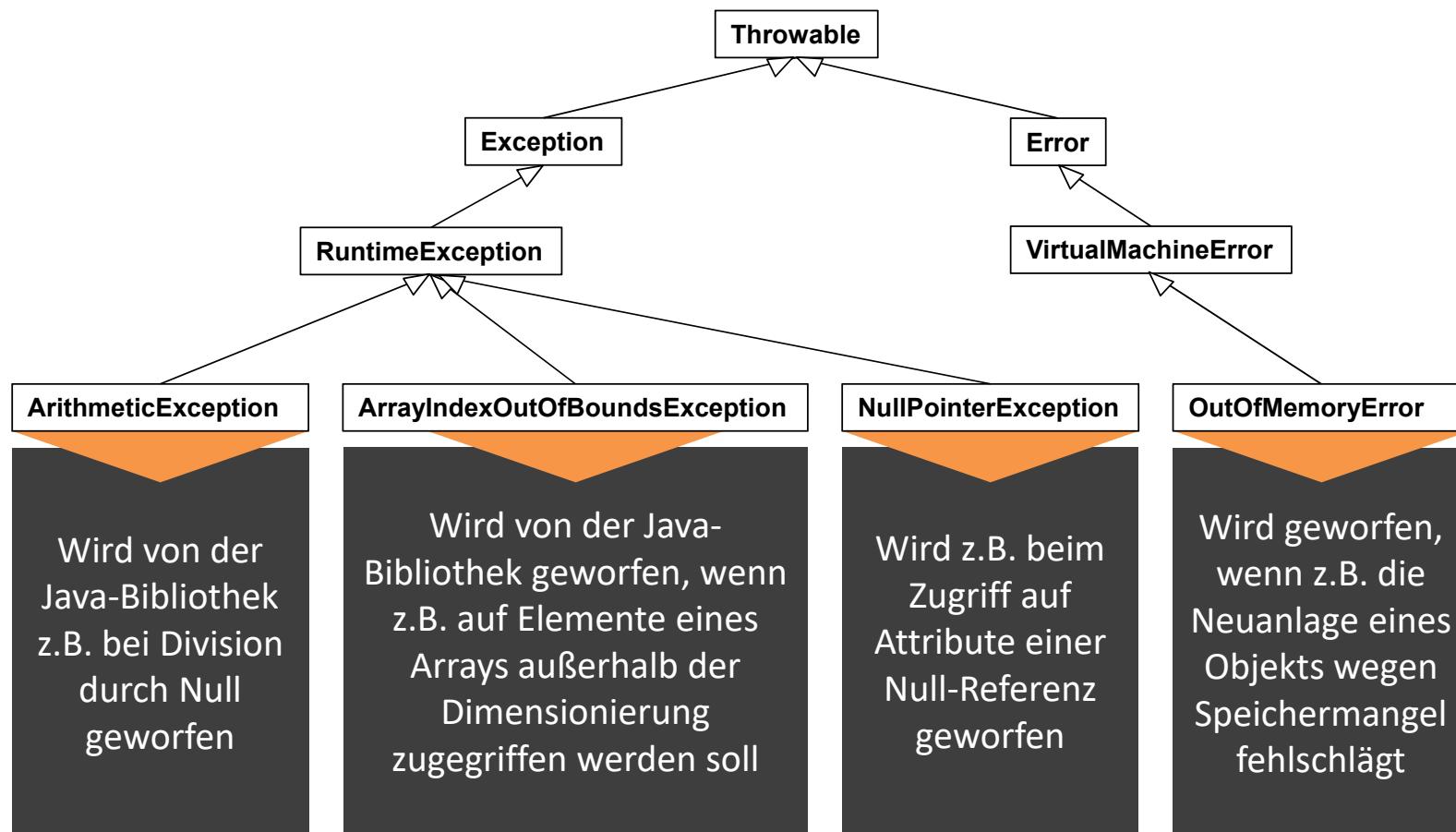
Hier wird der Fehler tatsächlich ausgelöst:
Es wird ein neues Exception-Objekt angelegt und über das **throw**-Schlüsselwort geworfen.

Exception-Klassen in `java.lang`

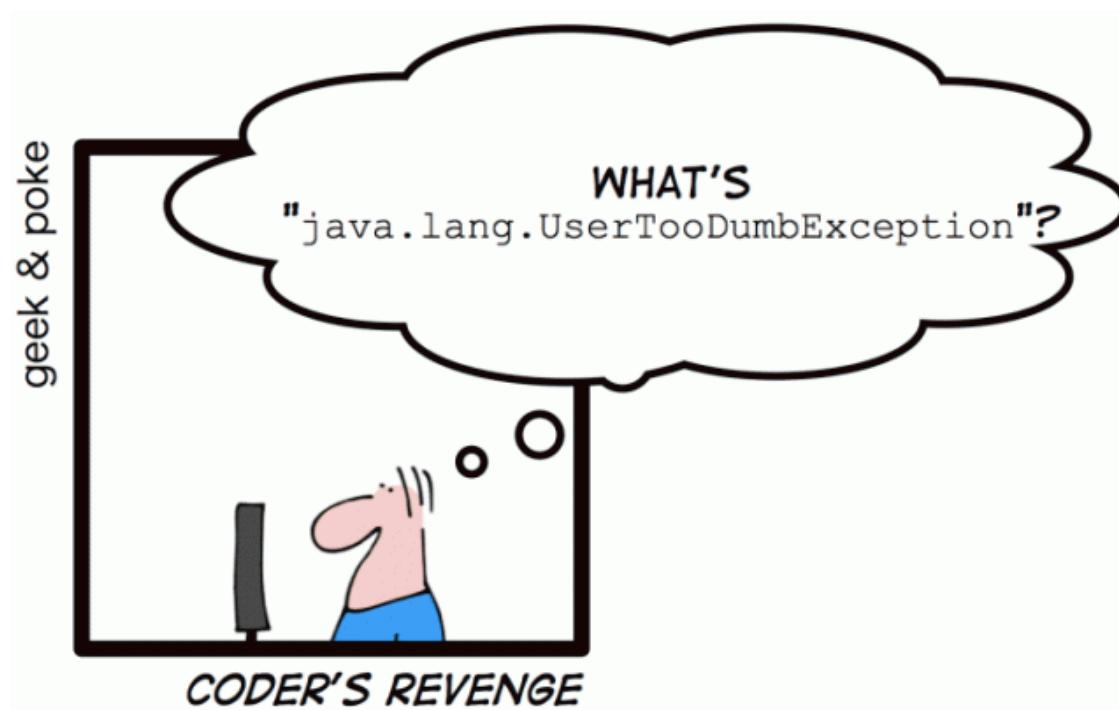
- Das Package `java.lang` liefert bereits eine große Auswahl von Exception-Klassen.
- Geworfen werden können Instanzen der Klasse `Throwable` und aller ihrer Unterklassen.
- Die Klasse `Error` und deren Unterklassen signalisieren fatale Fehler, die i.d.R. nicht behoben werden können.
- Die Klasse `Exception` und deren Unterklassen weisen auf Ausnahmesituationen hin, die im Programm behandelt werden können.



Einige Standard-Exceptions



Cartoon

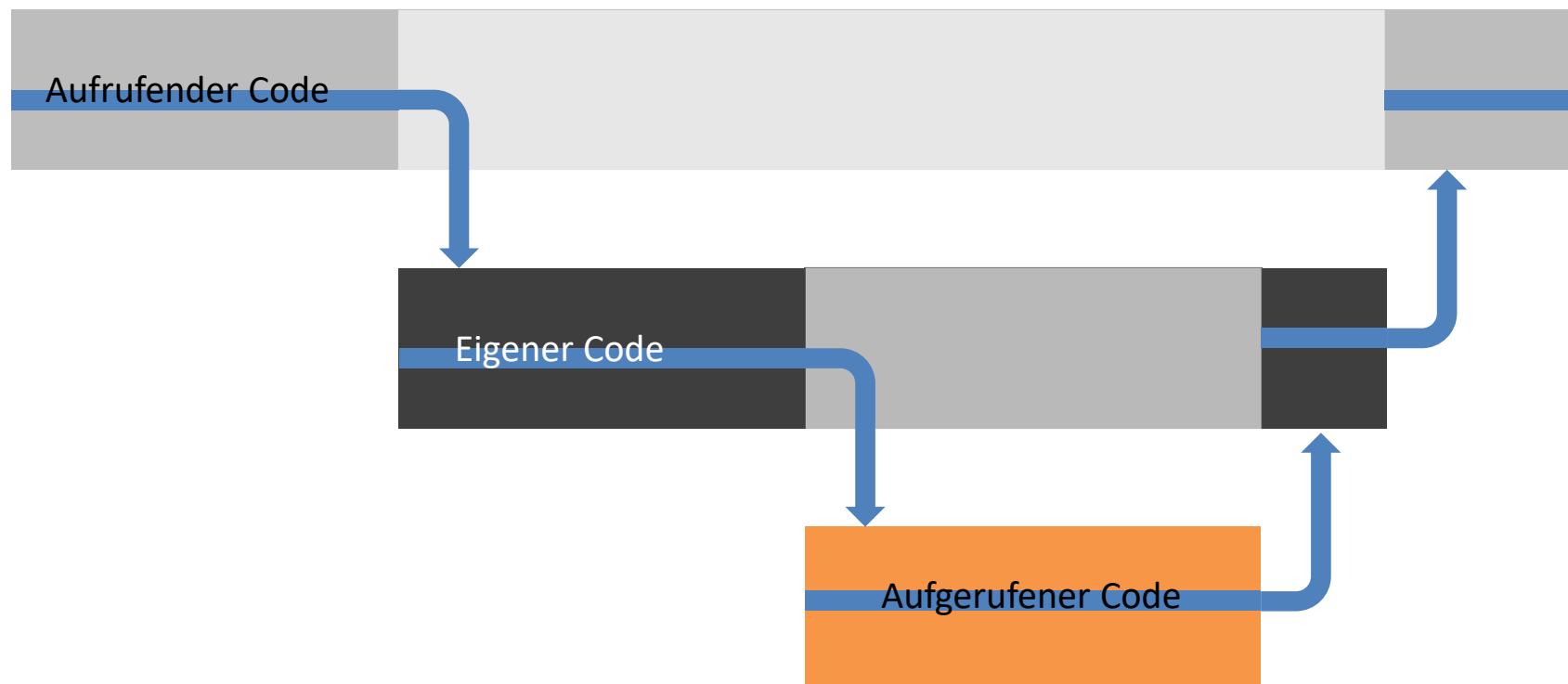


Anstatt der Objekte der Klasse Exception wirft man i.d.R.

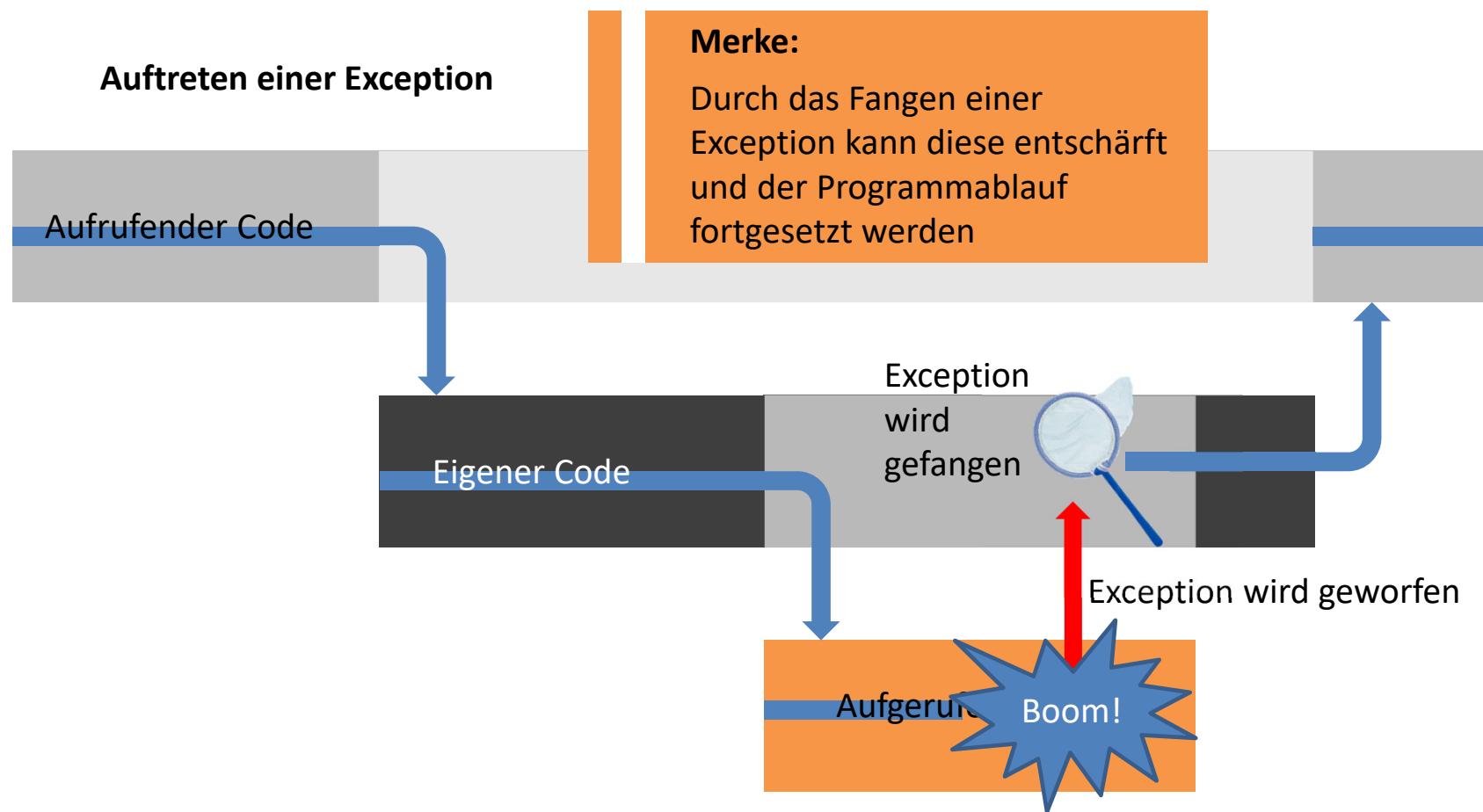
- ein Objekt von einer möglichst spezifischen Standardklasse
- oder von einer eigenen Exception-Klasse

Behandeln von Exceptions

Normaler Ablauf von verschachtelten Methodenaufrufen



Behandeln von Exceptions (I)



Das Fangen von Exceptions haben wir bereits gesehen:

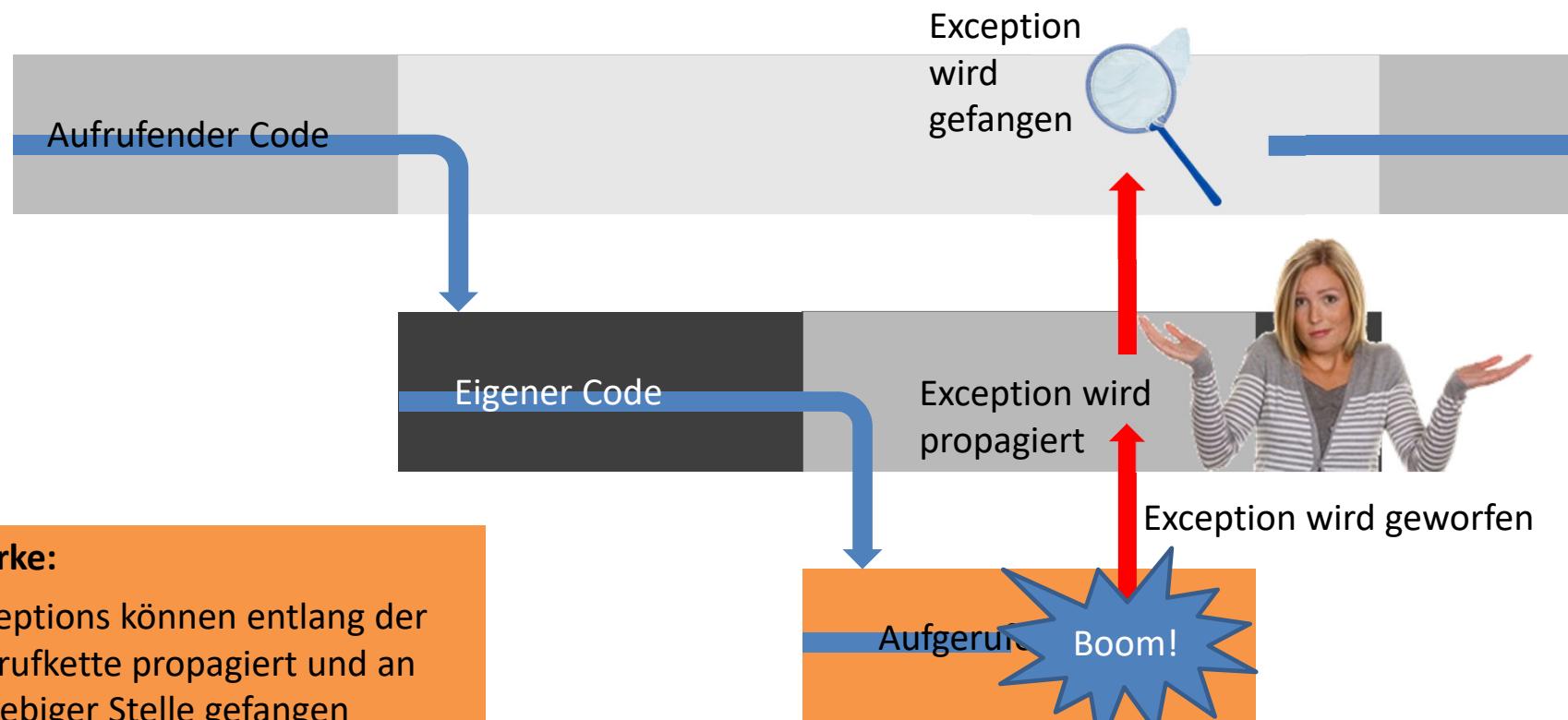
```
public static int readNumber() {  
    @SuppressWarnings("resource")  
    Scanner scanner = new Scanner(System.in);  
    do {  
        try {  
            String eingabe = scanner.nextLine();  
            int zahl = Integer.valueOf(eingabe);  
            return zahl;  
        }  
        catch (NumberFormatException e) {  
        }  
    }  
    while (true);  
  
    public static void main(String[] args) {  
        System.out.println("Die eingegebene Nummer war: " + readNumber());  
    }  
}
```

Unterdrückt die Warnung, wenn der Scanner nicht geschlossen wird.

Die Exception wird gefangen und der Fehler “beoben”.

Behandeln von Exceptions (II)

Auftreten einer Exception



Wir können uns auch dafür entscheiden die Exception zu propagieren:

```
public static int readNumber() throws NumberFormatException {  
    @SuppressWarnings("resource")  
    Scanner scanner = new Scanner(System.in);  
    String eingabe = scanner.nextLine();  
    int zahl = Integer.valueOf(eingabe); ← 2u  
    return zahl;  
}
```

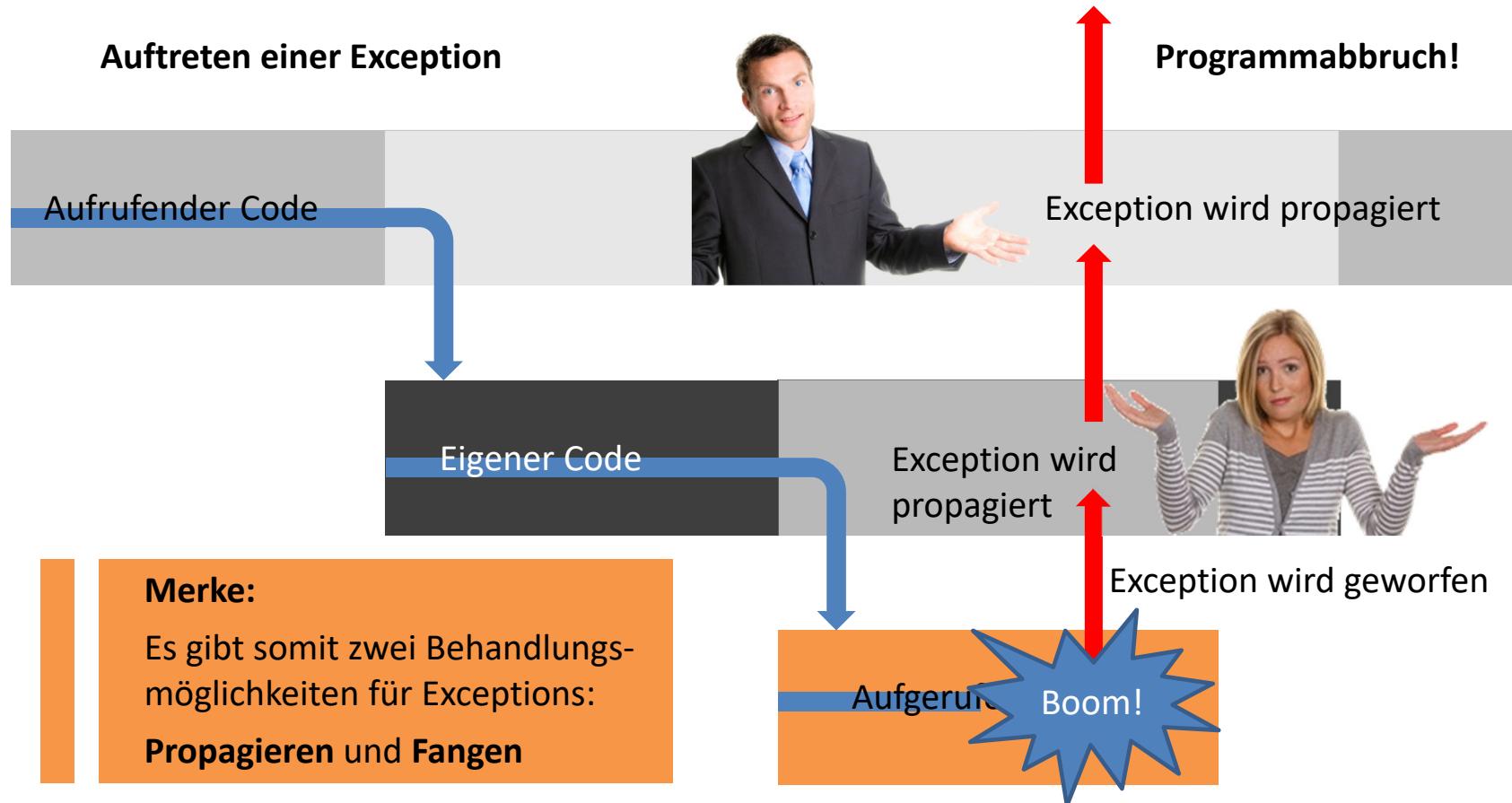
```
public static void main(String[] args) {  
    int zahl;  
    do {  
        try {  
            zahl = readNumber();  
            break;  
        }  
        catch (NumberFormatException e) {  
        }  
    }  
    while (true);  
    System.out.println("Die eingegebene Nummer war: " + zahl);  
}
```

Mit throws zeigen wir an:
In der Methode readNumber()
kann eine
NumberFormatException auftreten

Da die Methode eine auftretende
Exception nicht fängt, wird sie weiter
geworfen/propagiert.

Die Exception wird hier aufgefangen.

Behandeln von Exceptions (III)



Wir könnten sogar die Exception von der main-Methode aus propagieren:

```
public static int readNumber() throws NumberFormatException {  
    @SuppressWarnings("resource")  
    Scanner scanner = new Scanner(System.in);  
    String eingabe = scanner.nextLine();  
    int zahl = Integer.valueOf(eingabe); ← 2u  
    return zahl;  
}  
  
public static void main(String[] args) throws NumberFormatException {  
    System.out.println(readNumber());  
}
```

Allerdings kann jetzt nicht mehr auf den Fehler (eine falsche Eingabe) reagiert werden.

Wie fängt man eine Exception?



Im Fall einer Exception werden alle **catch**-Konstrukte von oben beginnend der Reihe nach Übereinstimmung überprüft.



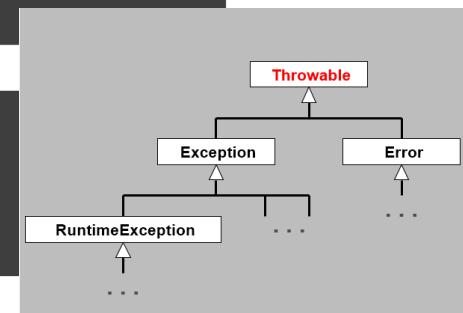
Übereinstimmung liegt dann vor, wenn es mit dem Exception-Typ des **catch**-Konstruktks **übereinstimmt** oder von diesem **abgeleitet** ist.



Oben in der Hierarchie befindliche Exception-Typen des bei **Throwable** beginnenden Klassenbaums müssen daher als letzte **catch**-Konstrukte aufgeführt werden.



Mit einem **catch**-Konstrukt vom Exception-Typ **Throwable** können alle Exceptions gefangen werden (sollte man i.d.R. nicht machen).



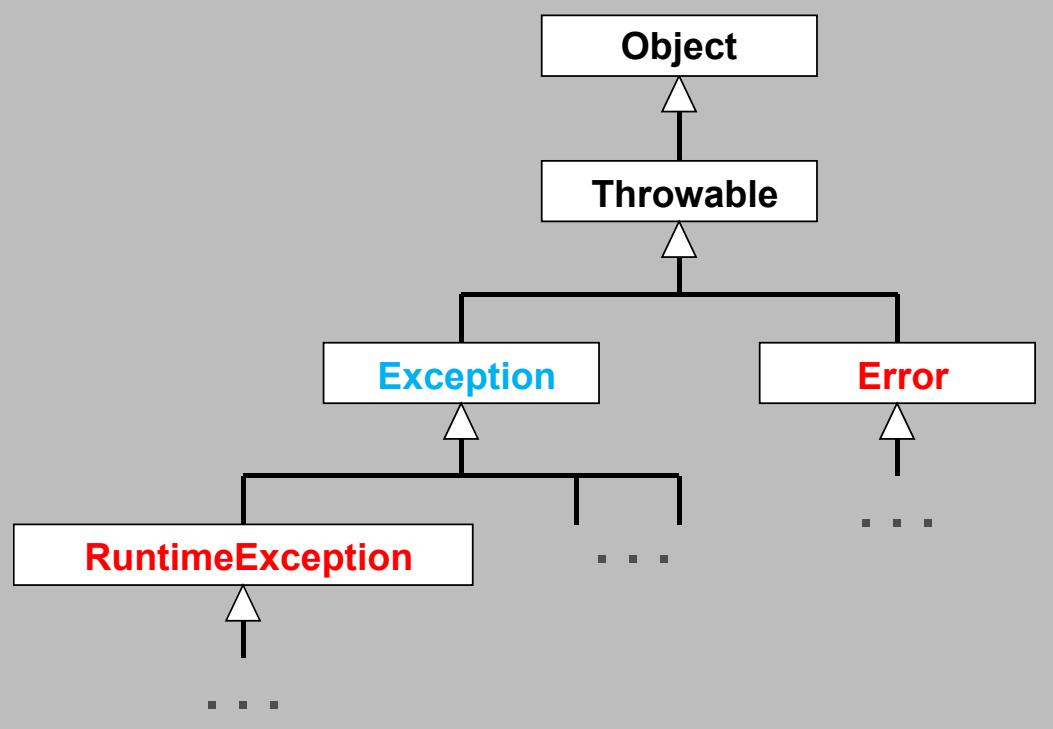
Es gibt grundsätzlich zwei verschiedene Arten von Exception:

Checked Exceptions

- sind alle Exception-Typen, die von der Klasse **Exception** abgeleitet sind, nicht aber von **RuntimeException**
- Behandlung (Propagieren oder Fangen) ist verpflichtend.

Unchecked Exceptions

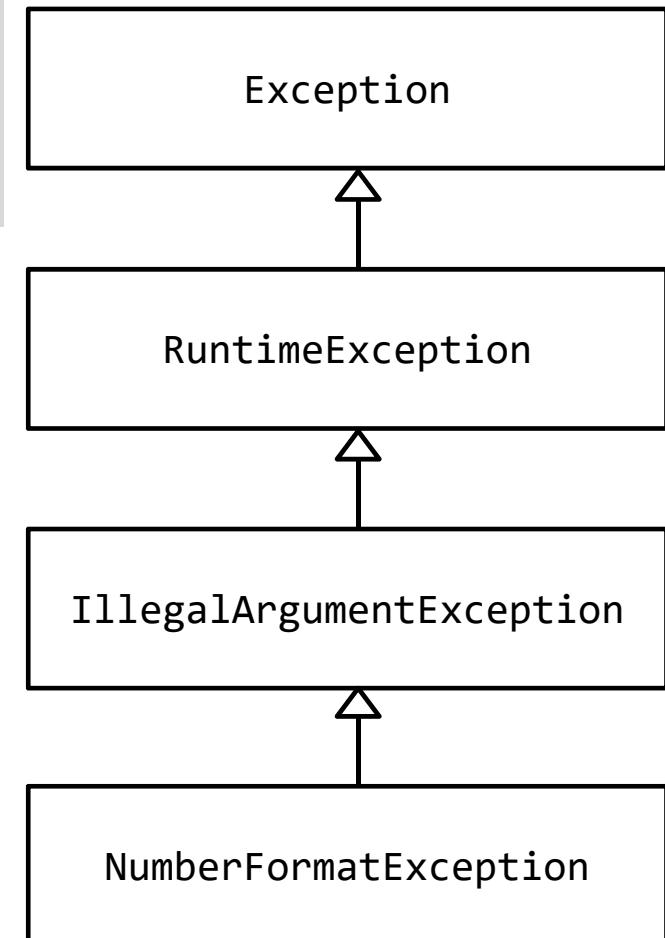
- sind alle Exception-Typen, die von den Klassen **Error** oder **RuntimeException** abgeleitet sind.
- Behandlung ist möglich, aber nicht verpflichtend.
- Unbehandelte Exceptions werden auch ohne throws-Anweisung propagiert.



```
public static int readNumber() throws NumberFormatException {  
    @SuppressWarnings("resource")  
    Scanner scanner = new Scanner(System.in);  
    String eingabe = scanner.nextLine();  
    int zahl = Integer.valueOf(eingabe);  
    return zahl;  
}
```

Da die NumberFormatException eine Unterklasse von RuntimeException ist, gehört sie zur Gruppe der unchecked Exceptions.

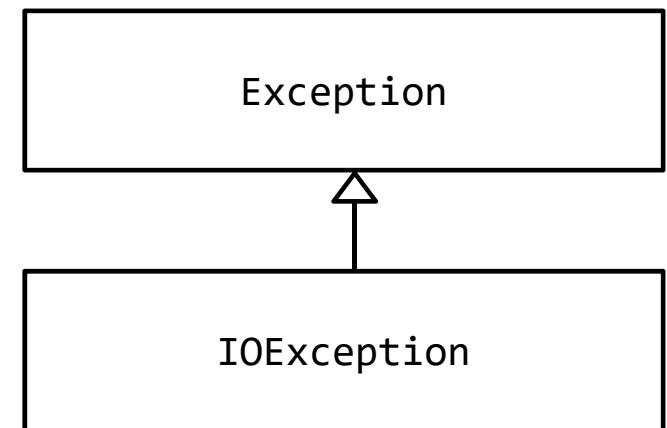
- Die Behandlung der Exception ist damit optional.
- Das explizite Propagieren der Exception kann entfallen.
- Die throws-Anweisung ist damit nicht notwendig.



Checked Exceptions

Wir kennen bereits das Propagieren von Exceptions vom Einlesen von Einzelzeichen.

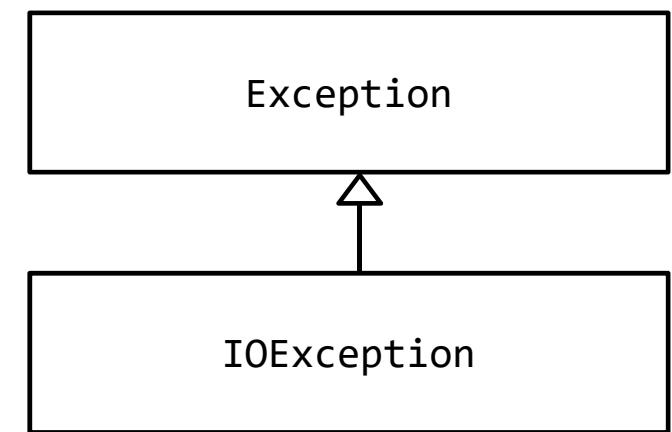
```
public static void main(String[] args) throws IOException
{
    char c = (char) System.in.read();
    System.out.println(c);
}
```



Die Methode `read()` wirft eine `IOException`.
Da `IOException` direkt von `Exception` abgeleitet ist,
ist `IOException` eine Checked Exception und muss
gefangen oder propagiert werden.

Wir können die Exception auch fangen:

```
public static void main(String[] args)
{
    try
    {
        char c = (char) System.in.read();
    }
    catch (IOException e)
    {
    }
}
```



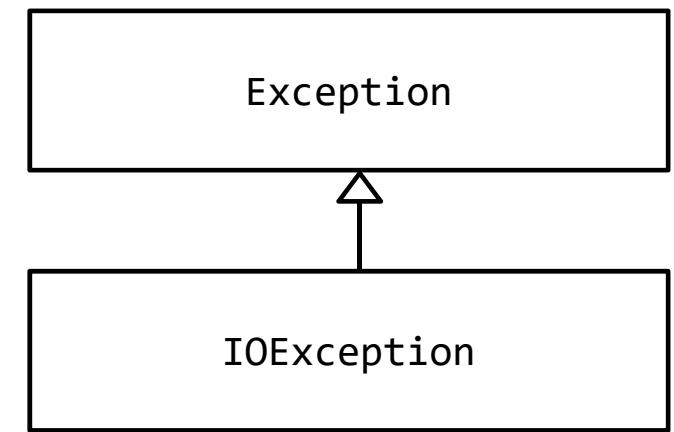
Allerdings kann vom Standard Input nur schwer ein Fehler erzeugt werden.

Ein Fehler ließe sich erzeugen, wenn der
Standard Input geschlossen wurde:

```
public static void main(String[] args)
{
    Scanner s = new Scanner(System.in);
    ...
    s.close();
    try
    {
        char c = (char) System.in.read();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

e.printStackTrace() gibt den
Methodenaufrufstack aus.

Anhand des Stacks Trace kann
man verfolgen, über welchen
Weg die Exception propagiert
wurde.



Wie sieht ein beispielhafter Methodenaufrufstack
(Stack Trace) bei einem auftretenden Fehler aus:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "afs"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.valueOf(Integer.java:766)
at exceptions.ScannerExceptionTest.testWithNextLineOnly(ScannerExceptionTest.java:75)
at exceptions.ScannerExceptionTest.main(ScannerExceptionTest.java:13)
```

Besonders interessant (neben dem Exception Namen und der Fehlermeldung in der ersten Zeile) sind i.d.R. zwei Zeilen im Stack Trace:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "afs"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.valueOf(Integer.java:766)
at exceptions.ScannerExceptionTest.testWithNextLineOnly(ScannerExceptionTest.java:75)
at exceptions.ScannerExceptionTest.main(ScannerExceptionTest.java:13)
```

1: Der Übergang zu unserem eigenen Quellcode.

Über dieser Zeile sind Methodenaufrufe von den Java-Standardklassen oder einer Ausführungsumgebung.

Besonders interessant (neben dem Exception-Namen und der Fehlermeldung in der ersten Zeile) sind i.d.R. zwei Zeilen im Stack Trace:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "afs"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.valueOf(Integer.java:766)
at exceptions.ScannerExceptionTest.testWithNextLineOnly(ScannerExceptionTest.java:75)
at exceptions.ScannerExceptionTest.main(ScannerExceptionTest.java:13)
```

2: Die Stelle, an der unser Quellcode endet.

Unter dieser Zeile endet der Stack Trace oder es befindet sich Methodenaufrufe einer Ausführungsumgebung.

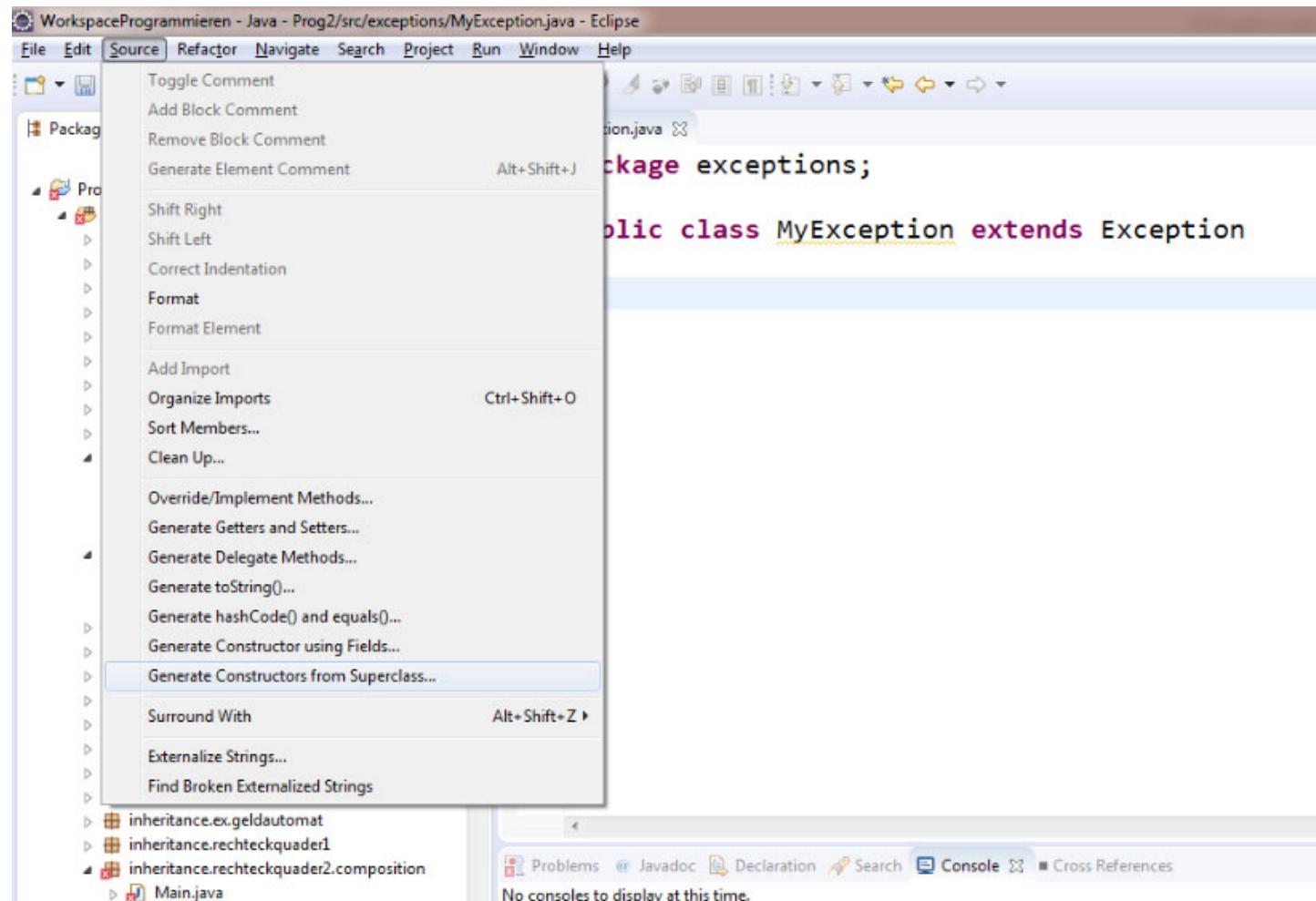
Eigene Exceptions

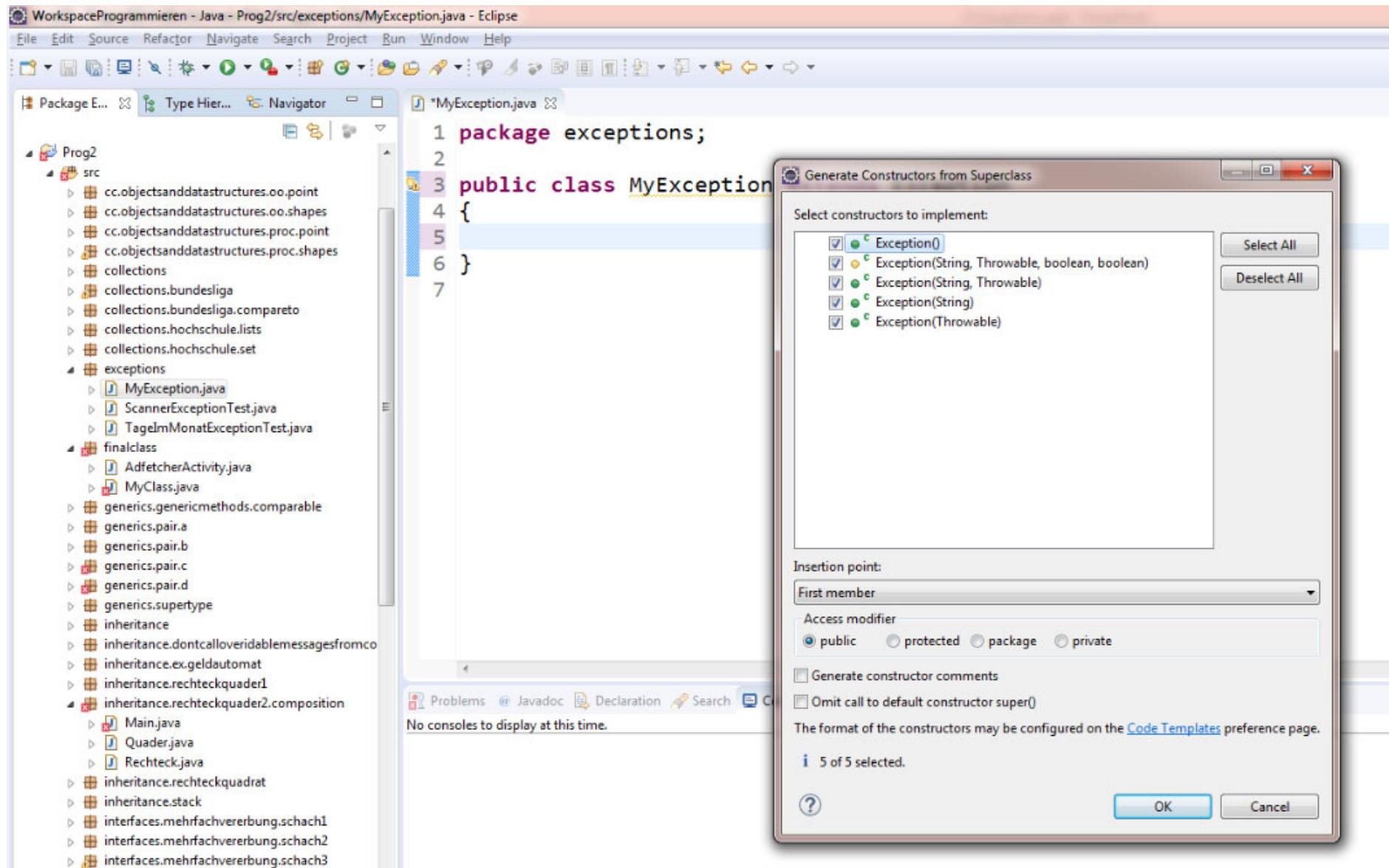
Eigene Exception-Typen

-  Neben den von Java vordefinierten Ausnahmetypen besteht die Möglichkeit eigene, speziell auf die Situation bzw. das Problem zugeschnittene Ausnahmetypen zu vereinbaren.
-  Ein selbstdefinierter Ausnahmetyp ist eine Klasse, die von der Klasse **Throwable** oder einer ihrer Subklassen (z.B. **Exception**) abgeleitet ist.
-  Durch entsprechendes Einhängen in die Vererbungshierarchie können sowohl checked als auch unchecked Exceptions selbst definiert werden.

```
J *MyException.java ✘
1 package exceptions;
2
3 public class MyException extends Exception
4 {
5
6 }
7
```

oder extends RuntimeException, wenn es
eine unchecked exception sein soll...





leinl

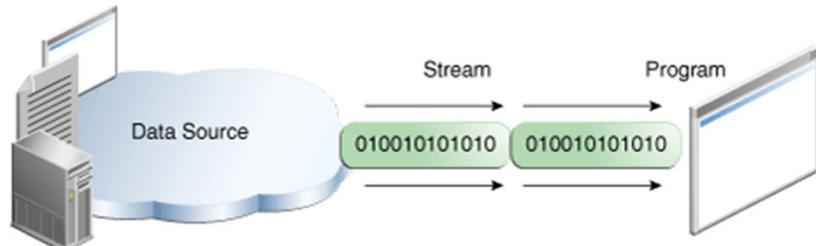
```
1 package exceptions;
2
3 public class MyException extends Exception
4 {
5     public MyException()
6     {
7         super();
8     }
9
10    public MyException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)
11    {
12        super(message, cause, enableSuppression, writableStackTrace);
13    }
14
15    public MyException(String message, Throwable cause)
16    {
17        super(message, cause);
18    }
19
20    public MyException(String message)
21    {
22        super(message);
23    }
24
25    public MyException(Throwable cause)
26    {
27        super(cause);
28    }
29 }
```

I.d.R. genügt es von der Klasse Exception oder
RuntimeException zu erben und alle
Konstruktoren der Oberklasse "zu erben".

I/O Streams

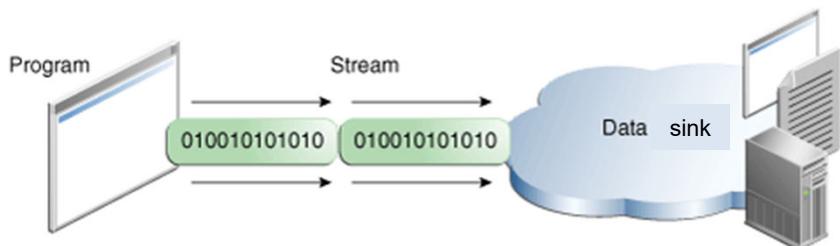
Was ist ein Stream?

- Ein Stream ist eine Abfolge von Daten.



source: <https://docs.oracle.com/javase/tutorial/essential/iostreams.html>

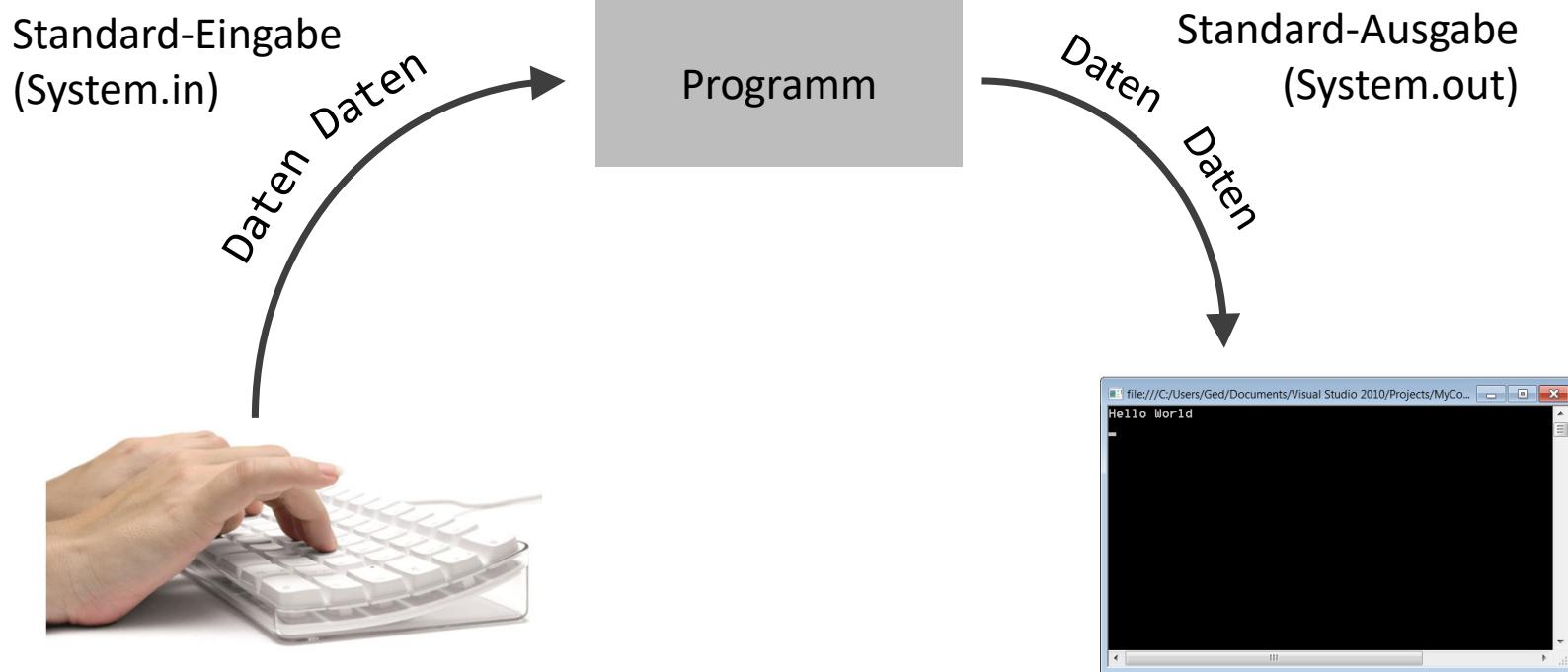
Ein Input Stream liest die Daten von einer Datenquelle in ein Programm.



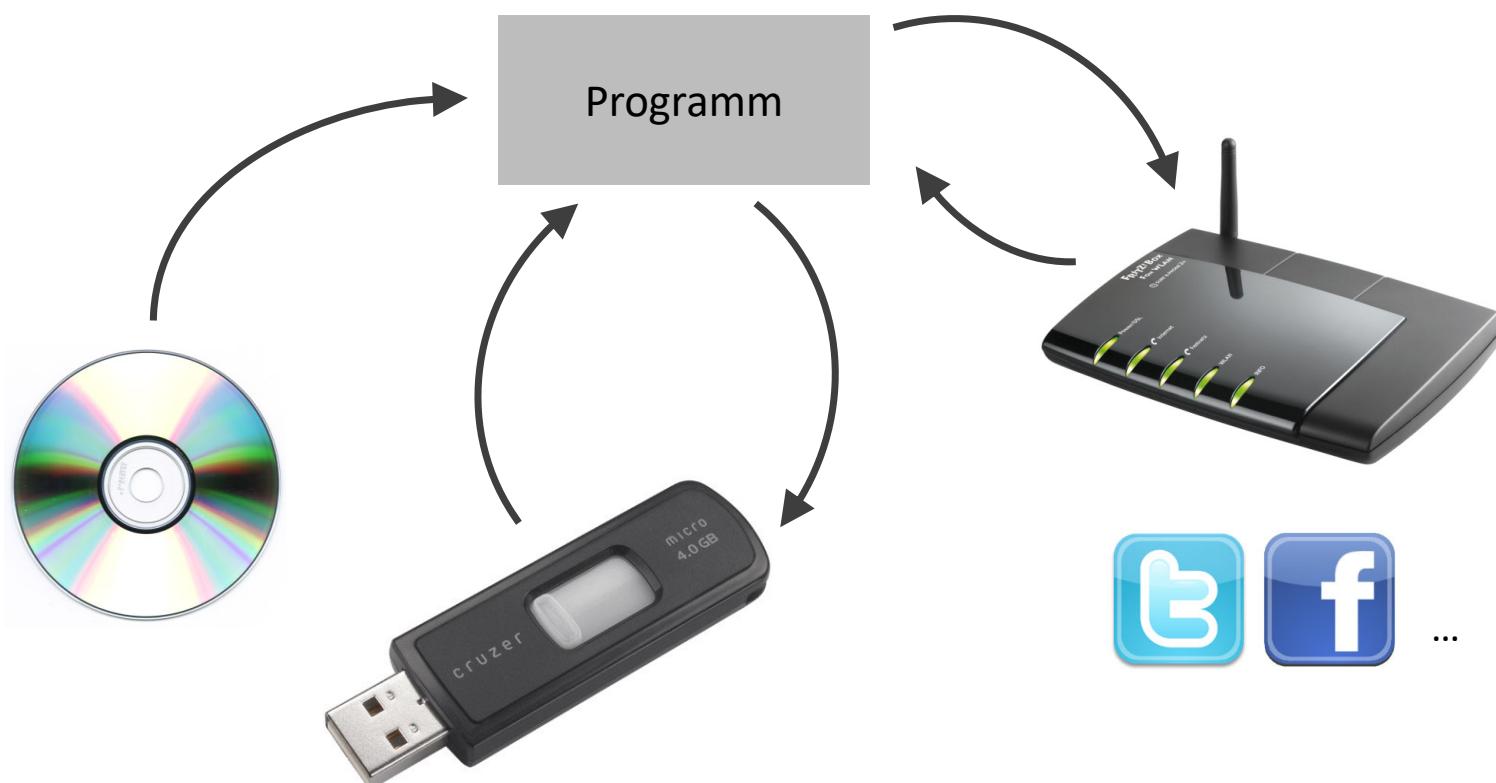
source: <https://docs.oracle.com/javase/tutorial/essential/iostreams.html>

Ein Output Stream schreibt Daten von einem Programm in eine Datensenke.

Was sind Streams?



Was sind Streams?

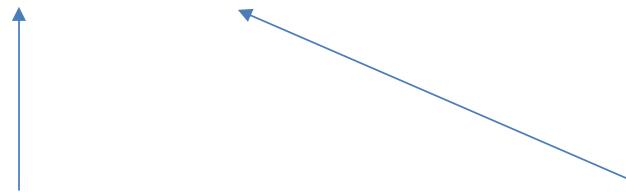


Eingabe und Ausgabe

- Streams lesen
 - von der Tastatur (Standardinput)
 - aus einer Datei
 - aus dem Hauptspeicher
 - von Sockets (Netzwerk)
- Streams schreiben
 - auf den Standardoutput (wird i.d.R. weitergeleitet auf Konsole/Terminal)
 - in eine Datei
 - in den Hauptspeicher
 - auf Sockets (Netzwerk)

Welche Arten von Daten werden durch Streams verarbeitet?

Textuelle Daten

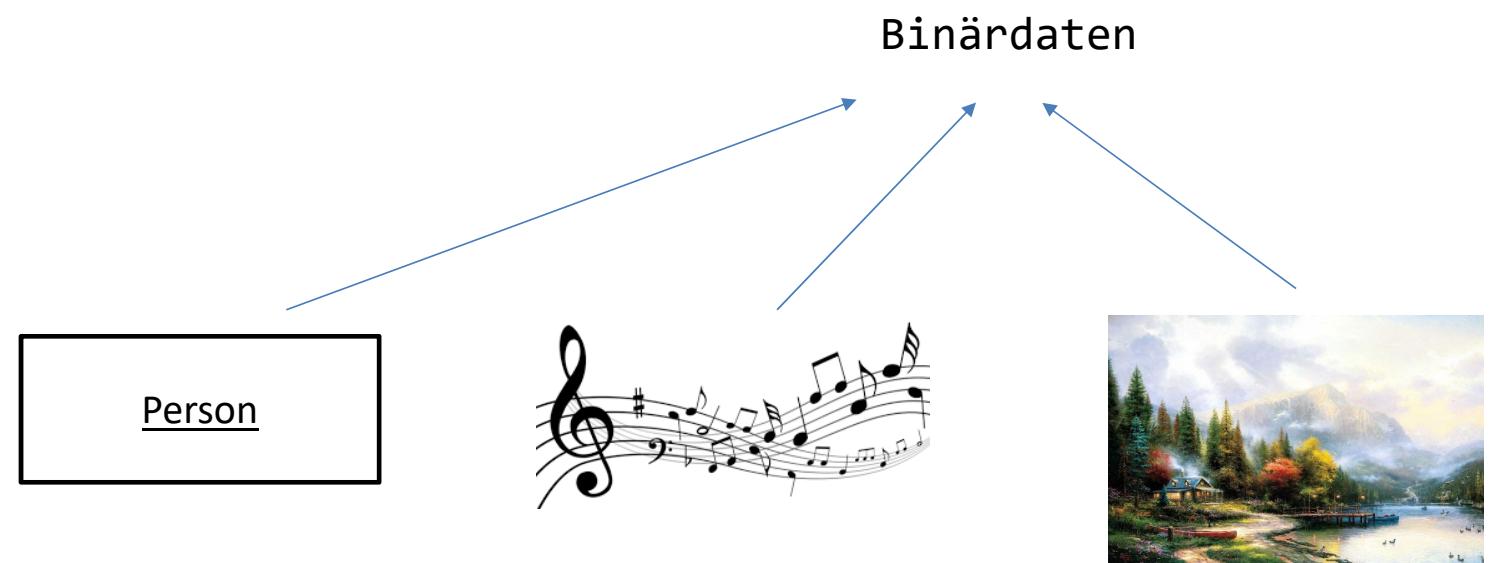


clean code (plural clean codes)

software code that is formatted in a modularized and systematic manner so that another coder can easily interpret or modify it.

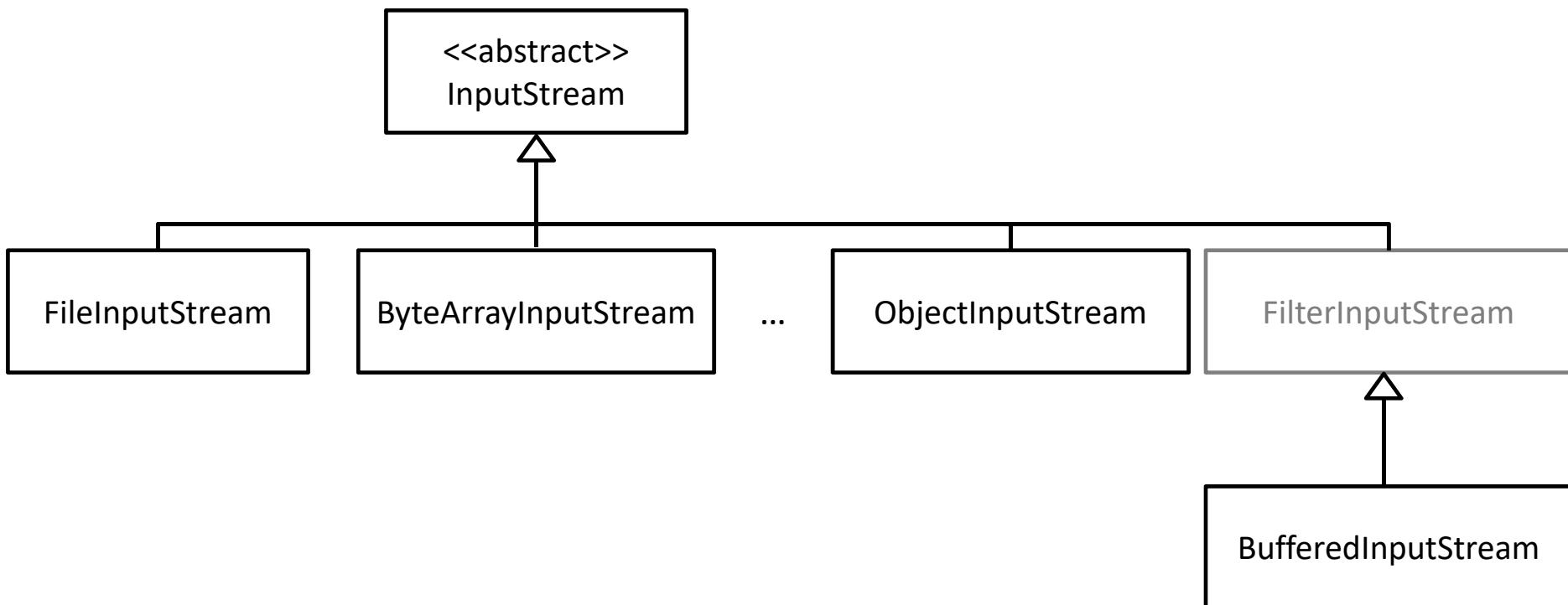
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ... version="3.0">
    <display-name>JSFPlatineShop</display-name>
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>
</web-app>
```

Welche Arten von Daten werden durch Streams verarbeitet?



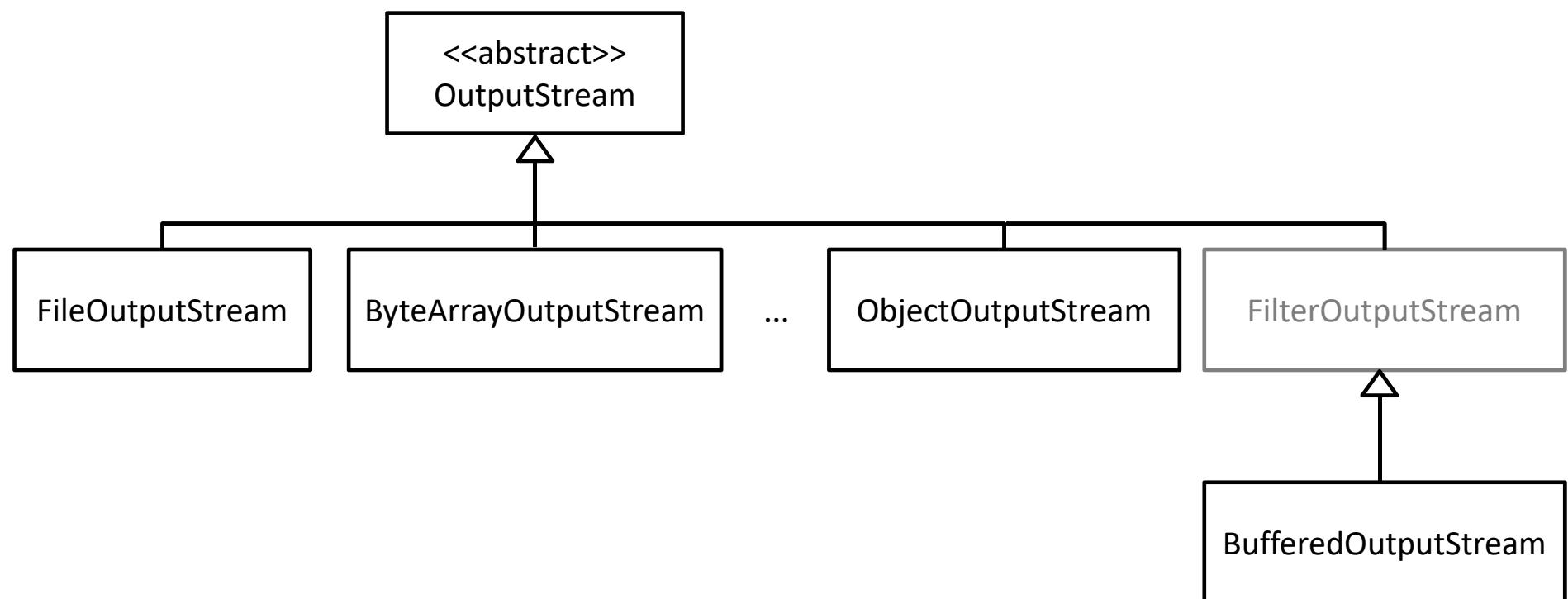
Binärdaten werden durch sog. **Byte Streams** verarbeitet.

Ein Ausschnitt der Input Streams aus der Java Klassenbibliothek:



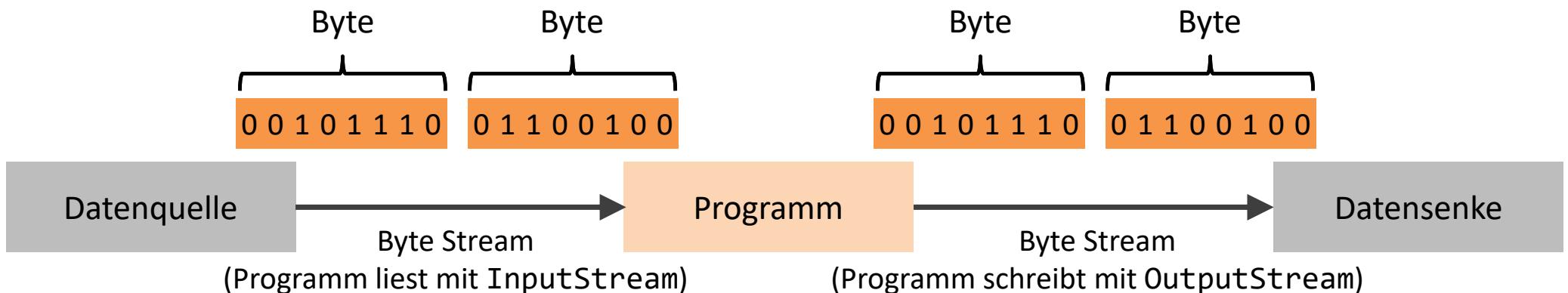
Binärdaten werden durch sog. **Byte Streams** verarbeitet.

Ein Ausschnitt der Output Streams aus der Java Klassenbibliothek.



Byte Streams

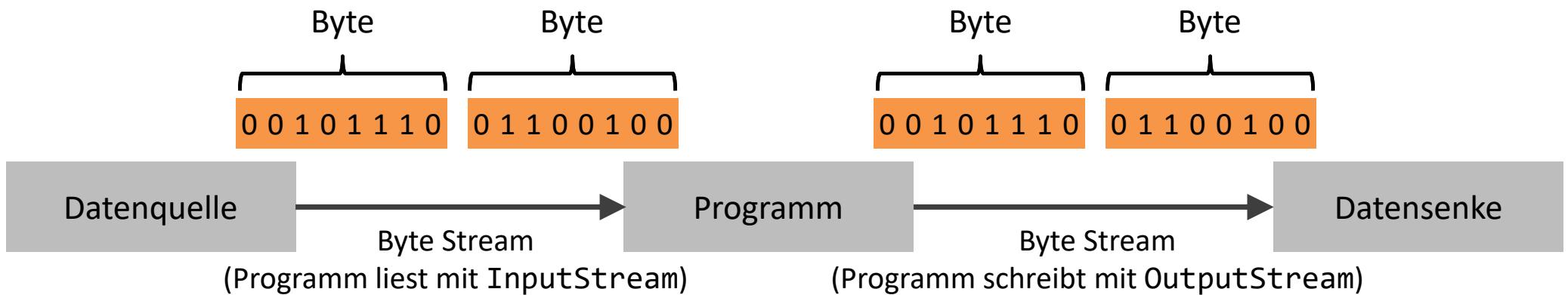
Byte Streams



```
public abstract class InputStream  
implements Closeable {  
  
    public abstract int read() throws IOException;  
    public int read(byte b[]) throws IOException { ... }  
    public int read(byte b[], int off, int len)  
        throws IOException { ... }  
  
    public int available() throws IOException { ... }  
  
    public void close() throws IOException {}  
}
```

```
public abstract class OutputStream  
implements Closeable, Flushable {  
  
    public abstract void write(int b)  
        throws IOException;  
    public void write(byte b[])  
        throws IOException { ... }  
    public void write(byte b[], int off, int len)  
        throws IOException { ... }  
  
    public void flush() throws IOException {}  
    public void close() throws IOException {}  
}
```

Byte Streams



```
public abstract class InputStream
    implements Closeable {

    public abstract int read() throws IOException;
}

read() liefert das nächste Byte zurück oder -1,
wenn das Ende des Streams erreicht wurde.
```

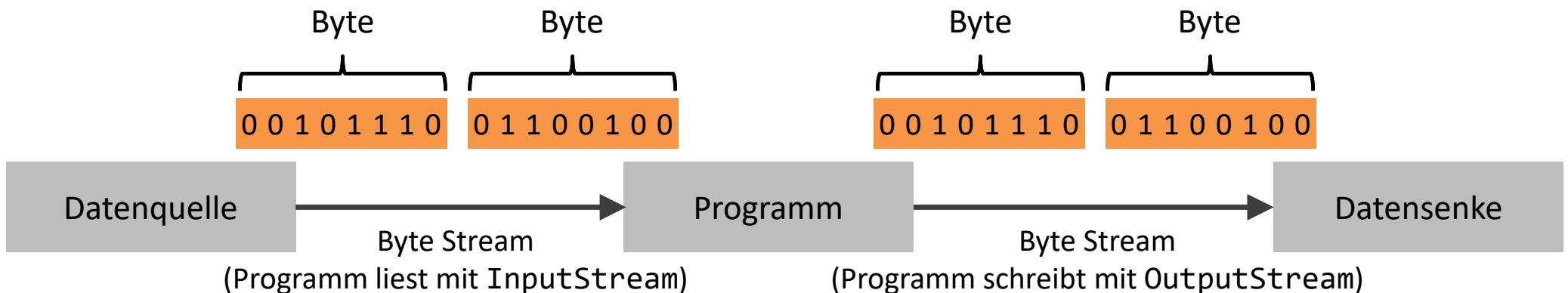
```
public abstract class OutputStream
    implements Closeable, Flushable {

    public abstract void write(int b)
        throws IOException;
    public void write(byte b[])
        throws IOException { ... }
    public void write(byte b[], int off, int len)
        throws IOException { ... }

    public void flush() throws IOException {}
    public void close() throws IOException {}

}
```

Byte Streams



```
public abstract class InputStream
    implements Closeable {

    public abstract int read() throws IOException;
    public int read(byte b[]) throws IOException { ... }

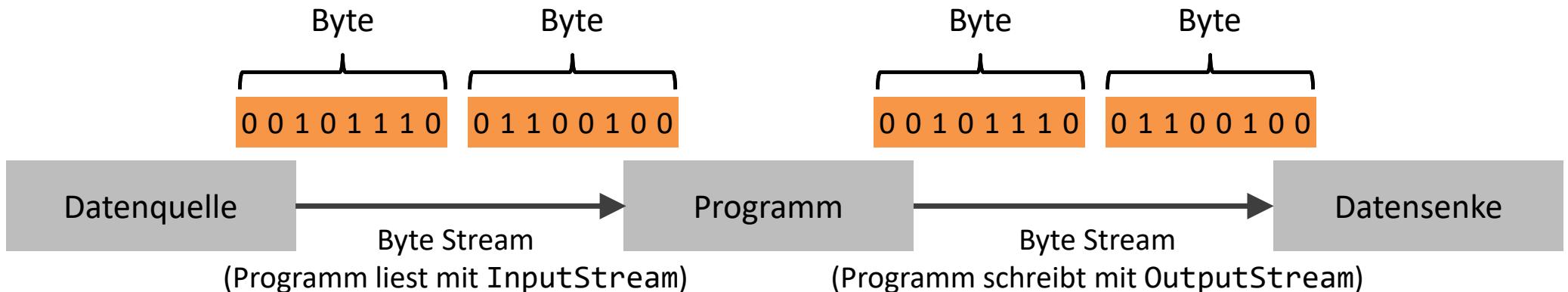
    read(byte[]) schreibt die gelesenen Bytes in das
    übergebene byte[] und liefert die Anzahl der
    gelesenen Bytes zurück (oder -1, wenn das Ende
    des Streams erreicht wurde).
}
```

```
public abstract class OutputStream
    implements Closeable, Flushable {

    public abstract void write(int b)
        throws IOException;
    public void write(byte b[])
        throws IOException { ... }
    public void write(byte b[], int off, int len)
        throws IOException { ... }

    public void flush() throws IOException {}
    public void close() throws IOException {}
}
```

Byte Streams



```
public abstract class InputStream
    implements Closeable {

    public abstract int read() throws IOException;
public int read(byte b[]) throws IOException { ... }
    public int read(byte b[], int off, int len)
        throws IOException { ... }

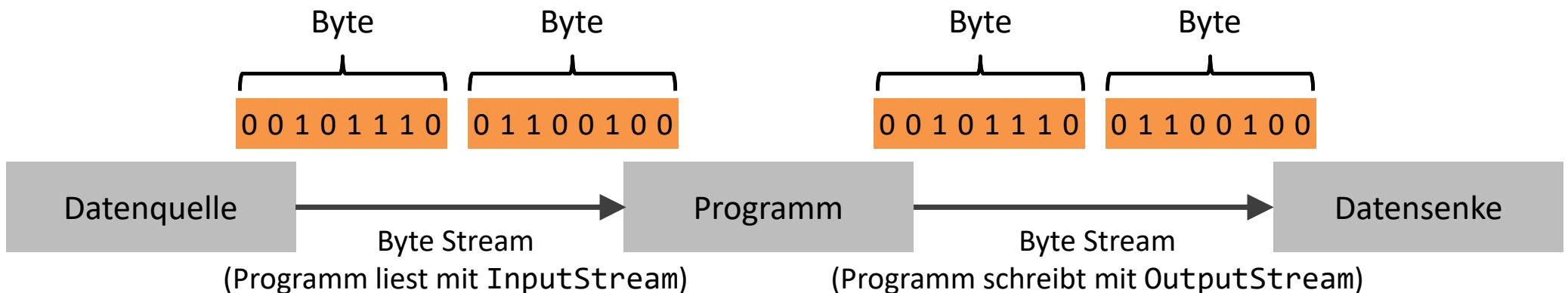
    public int available() throws IOException { ... }

    public void close() throws IOException {}
}
```

```
public abstract class OutputStream
    implements Closeable, Flushable {

    public abstract void write(int b)
        throws IOException;
write() schreibt ein byte in den Stream
}
```

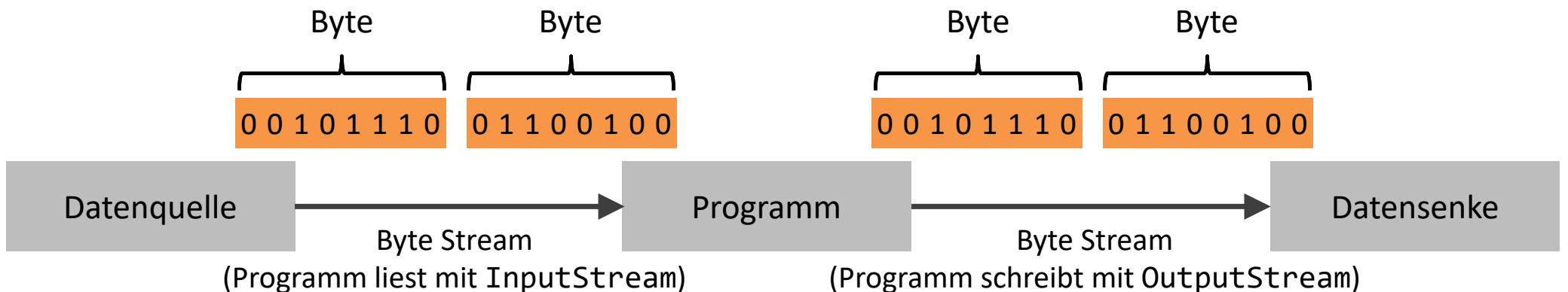
Byte Streams



```
public abstract class InputStream  
implements Closeable {  
  
    public abstract int read() throws IOException;  
    public int read(byte b[]) throws IOException { ... }  
    public int read(byte b[], int off, int len)  
        throws IOException { ... }  
  
    public int available() throws IOException { ... }  
  
    public void close() throws IOException {}  
}
```

```
public abstract class OutputStream  
implements Closeable, Flushable {  
  
    public abstract void write(int b)  
        throws IOException;  
    public void write(byte b[])  
        throws IOException { ... }  
  
    write(byte[]) schreibt das komplette byte[] in  
    den Stream  
}
```

Byte Streams



```
public abstract class InputStream
    implements Closeable {

    public abstract int read() throws IOException;
    public int read(byte b[]) throws IOException { ... }
    public int read(byte b[], int off, int len)
        throws IOException { ... }

    public int available() throws IOException { ... }

    public void close() throws IOException {}
}
```

```
public abstract class OutputStream
    implements Closeable, Flushable {

    public abstract void write(int b)
        throws IOException;
    public void write(byte b[])
        throws IOException { ... }
    public void write(byte b[], int off, int len)
        throws IOException { ... }

    write(byte[], int off, int len) schreibt ab
    der Stelle off len bytes des byte[] in den Stream
}
```

InputStream

```
int read() throws IOException  
int read(byte b[]) throws IOException  
int read(byte b[], int off, int len) throws IOException  
int available() throws IOException  
void close() throws IOException
```

OutputStream

```
void write(int b) throws IOException  
void write(byte b[]) throws IOException  
void write(byte b[], int off, int len) throws IOException  
void flush() throws IOException  
void close() throws IOException
```

Datei

Mit einer abstrakten Klasse (`OutputStream`/`InputStream`) kann man nicht viel tun...

Daher ein Beispiel mit konkreten Klassen:
Bilddatei kopieren



(Beim Öffnen einer Datei wird i.d.R. vom Betriebssystem ein Handle vergeben.)

Zunächst öffnen wir eine Datei!

```
public class DateiKopieren
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis = new FileInputStream("pic.jpg");
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        ...
    }
}
```

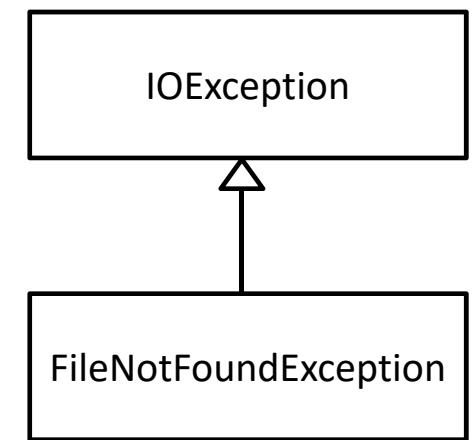
Die Datei liegt im lokalen Verzeichnis
(bzw. im Eclipse Projektordner) und
heißt pic.jpg.

Beim Öffnen der Datei **muss** eine
FileNotFoundException abgefangen
werden.

```

public class DateiKopieren
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis = new FileInputStream("pic.jpg");
            FileOutputStream fos = new FileOutputStream("copy.jpg");
            int b;
            do
            {
                b = fis.read();
                if (b != -1) fos.write(b);
            }
            while (b != -1);
            fis.close();
            fos.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

```



read, write und close werfen eine IOException.
Da FileNotFoundException von IOException abgeleitet ist,
genügt es zunächst, IOExceptions zu fangen.

Allerdings kann so nicht direkt unterschieden werden,
welche Exception aufgetreten ist.

```
public class DateiKopieren
{
    public static void main(String[] args)
    {
        try
        {
            InputStream fis = new FileInputStream("pic.jpg");
            OutputStream fos = new FileOutputStream("copy.jpg");
            int b;
            do
            {
                b = fis.read();
                if (b != -1) fos.write(b);
            }
            while (b != -1);
            fis.close();
            fos.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Wir benutzen keine FileInputStream- oder FileOutputStream-spezifischen Methoden.

Bisher haben wir keine Fehlerbehandlung:

Bei einem Fehler in der Verarbeitung werden die Streams nicht geschlossen!

Ein Betriebssystem kann i.d.R. nur eine gewisse Anzahl an offenen Netzwerkverbindungen und Dateien gleichzeitig verarbeiten.

Bleiben diese dauerhaft geöffnet, kann das Betriebssystem ggf. keine Dateien (und Netzwerkverbindungen) mehr öffnen.

```
public static void copyFileWithErrorHandling() {  
  
    try {  
        InputStream fis = new FileInputStream("pic.jpg");  
        OutputStream fos = new FileOutputStream("copy.jpg");  
        int b;  
        do {  
            b = fis.read();  
            if (b != -1) fos.write(b);  
        }  
        while (b != -1);  
    }  
    catch (IOException e) {  
        e.printStackTrace();  
    }  
    finally {  
  
        fis.close();  
        fos.close();  
    }  
}
```

Wir schließen die Streams in einem **finally**-Block.

Anweisungen im **finally**-Block werden auf jeden Fall ausgeführt, egal ob eine Exception geworfen wird oder nicht.

momentan sehen wir noch die Fehlermeldung:
fis cannot be resolved



```
public static void copyFileWithErrorHandling() {  
    InputStream fis = null;  
    OutputStream fos = null;  
  
    try {  
        fis = new FileInputStream("pic.jpg");  
        fos = new FileOutputStream("copy.jpg");  
        int b;  
        do {  
            b = fis.read();  
            if (b != -1) fos.write(b);  
        }  
        while (b != -1);  
    }  
    catch (IOException e) {  
        e.printStackTrace();  
    }  
    finally {  
        fis.close();  
        fos.close();  
    }  
}
```

fis.close() könnte ebenfalls eine Exception auslösen

Eine sinnvolle Fehlerbehandlung können wir an dieser Stelle i.d.R.
nicht durchführen!

```
public static void copyFileWithErrorHandling() {  
    InputStream fis = null;  
    OutputStream fos = null;  
  
    try {  
        fis = new FileInputStream("pic.jpg");  
        fos = new FileOutputStream("copy.jpg");  
        int b;  
        do {  
            b = fis.read();  
            if (b != -1) fos.write(b);  
        }  
        while (b != -1);  
    }  
    catch (IOException e) {  
        e.printStackTrace();  
    }  
    finally {  
        try {  
            if (fis != null) fis.close();  
            if (fos != null) fos.close();  
        }  
        catch(IOException e) {  
        }  
    }  
}
```

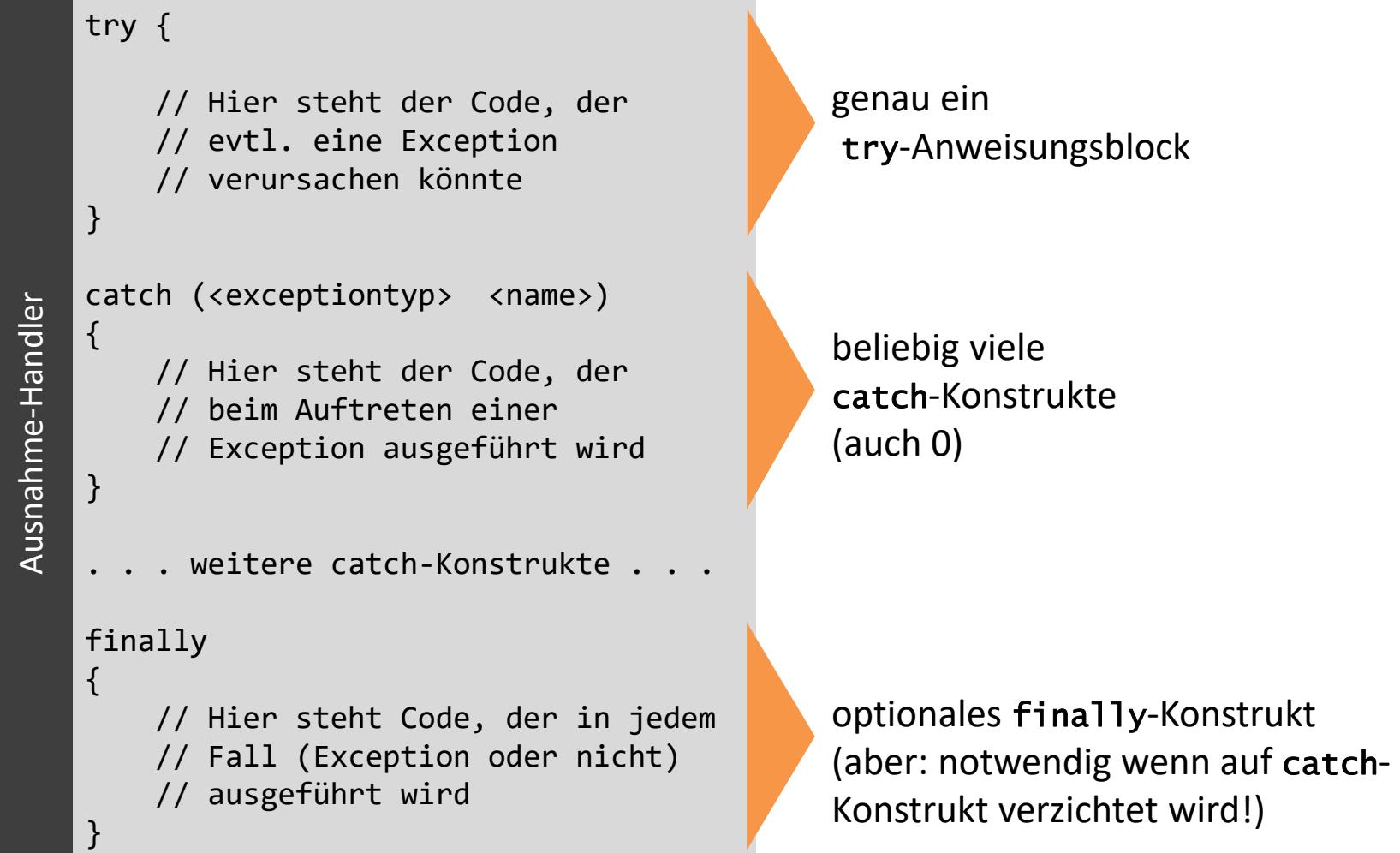
Ist die Fehlerbehandlung so in Ordnung?

Was passiert, wenn fis.close() eine Exception wirft?
Dann wird fos.close() nicht ausgeführt.

```
public static void copyFileWithErrorHandling() {  
    InputStream fis = null;  
    OutputStream fos = null;  
    try {  
        fis = new FileInputStream("pic.jpg");  
        fos = new FileOutputStream("copy.jpg");  
        int b;  
        do {  
            b = fis.read();  
            if (b != -1) fos.write(b);  
        }  
        while (b != -1);  
    }  
    catch (IOException e) {  
        e.printStackTrace();  
    }  
    finally {  
        try {  
            if (fis != null) fis.close();  
        }  
        catch (IOException e) {}  
        try {  
            if (fos != null) fos.close();  
        }  
        catch (IOException e) {}  
    }  
}
```

Jetzt werden der FileInputStream und der FileOutputStream unabhängig voneinander geschlossen.

Wie fängt man eine Exception?



try-with-resources statement

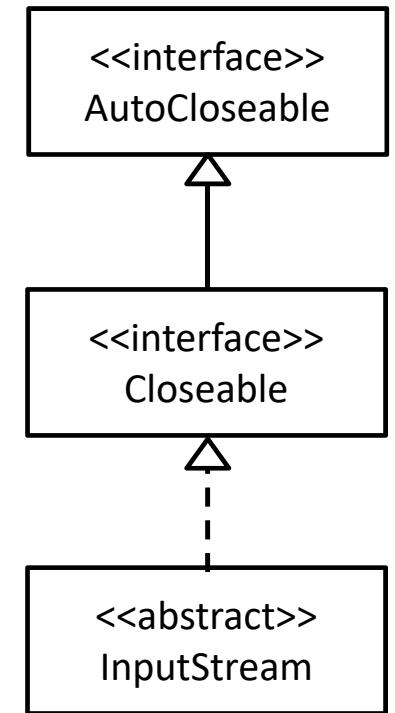
Das Fehlerhandling ist beim Verwenden mehrerer Streams aufwändig und selbst fehleranfällig.

Daher wurde in Java 7 das
try-with-resources statement eingeführt.

```

public static void copyFileWithTryWithResources()
{
    try(InputStream fis = new FileInputStream("pic.jpg");
        OutputStream fos = new FileOutputStream("copy.jpg");)
    {
        int b;
        do
        {
            b = fis.read();
            if (b != -1) fos.write(b);
        }
        while (b != -1);
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

```



try-with-resources statement erwartet in den Klammern die Deklaration von Objekten, die das AutoCloseable Interface implementieren.

Wir können den Code noch verbessern,
in dem wir das Kopieren auslagern.

```
private static void copy(InputStream is, OutputStream os) throws IOException
{
    int b;
    do
    {
        b = is.read();
        if (b != -1) os.write(b);
    }
    while (b != -1);
}

public static void copyFileWithErrorHandling3Modularized()
{
    try (InputStream fis = new FileInputStream("pic.jpg");
         OutputStream fos = new FileOutputStream("copy.jpg");)
    {
        copy(fis, fos);
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

Wiederverwendbar!

Noch ein Beispiel mit konkreten Klassen: Download und Speichern eines Bildes

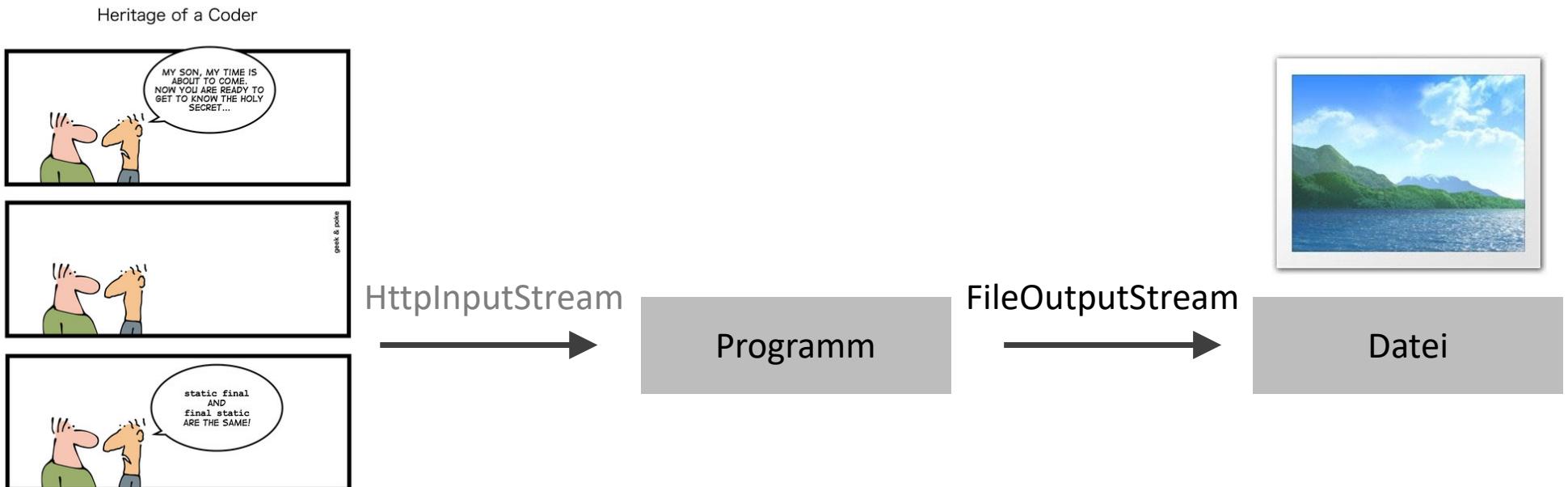


image URL: <http://static1.squarespace.com/static/518f5d62e4b075248d6a3f90/t/519bef92e4b02745db13b165/1369173914913/eol.jpg>

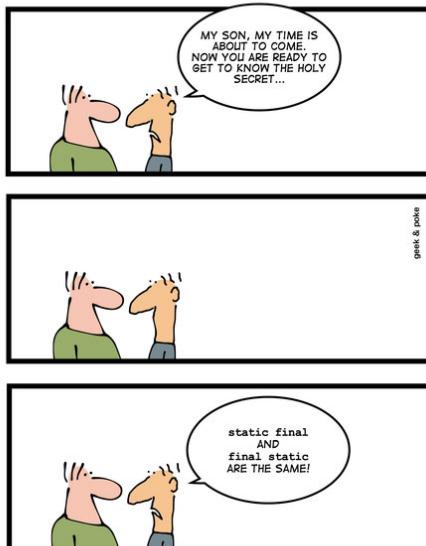
Wir verbessern zunächst die copy-Methode!

```

static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b);
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}

```

Heritage of a Coder



HttpInputStream

FileOutputStream

Programm

Datei



```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

read liest im ersten Durchgang 5 Bytes

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

Inhalt neu erstellte Datei:

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

read liest im ersten Durchgang 5 Bytes

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

Abcde

Inhalt neu erstellte Datei:

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

Wenn das Dateiende nicht erreicht wurde ($n \neq -1$), schreibe n Bytes ($n == 5$) in die Datei.

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

Abcde

Inhalt neu erstellte Datei:

Abcde

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

Die Netzwerkverbindung ist schlecht. read liest nur 3 Bytes.

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

Abcde

Inhalt neu erstellte Datei:

Abcde

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

Die Netzwerkverbindung ist schlecht. read liest nur 3 Bytes.

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

Fghde

Inhalt neu erstellte Datei:

Abcde

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

Wenn das Dateiende nicht erreicht wurde ($n \neq -1$), schreibe n Bytes ($n==3$) in die Datei.

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

Fghde

Inhalt neu erstellte Datei:

Abcde
Fgh

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

read liest 5 Bytes

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

Fghde

Inhalt neu erstellte Datei:

Abcde
Fgh

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

read liest 5 Bytes

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

ijklm

Inhalt neu erstellte Datei:

Abcde
Fgh

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

Wenn das Dateiende nicht erreicht wurde ($n \neq -1$), schreibe n Bytes ($n == 5$) in die Datei.

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

ijklm

Inhalt neu erstellte Datei:

Abcde
Fgh

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

Wenn das Dateiende nicht erreicht wurde ($n \neq -1$), schreibe n Bytes ($n == 5$) in die Datei.

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

ijklm

Inhalt neu erstellte Datei:

Abcde
Fghij
Klm

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

read liest das verbleibende Byte

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

njKlm

Inhalt neu erstellte Datei:

Abcde
Fghij
Klm

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

Wenn das Dateiende nicht erreicht wurde ($n \neq -1$), schreibe n Bytes ($n \geq 1$) in die Datei.

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

njKlm

Inhalt neu erstellte Datei:

Abcde
Fghij
Klmn

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[5];
    int n;
    do
    {
        n = is.read(b); //kann maximal 5 bytes lesen
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}
```

read liest eine -1, da wir am Dateiende angekommen sind und die Netzwerkverbindung geschlossen wurde.
Die Schleife endet.

Bilddateiinhalt:

Abcde
Fghij
Klmn

Inhalt Byte-Array b

njKlm

Inhalt neu erstellte Datei:

Abcde
Fghij
Klmn

Wir verwenden die bessere copy-Methode in
einem Programm zum Bilddateidownload!

```
static void copyImproved(InputStream is, OutputStream os) throws IOException
{
    byte[] b = new byte[4096];
    int n;
    do
    {
        n = is.read(b);
        if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
}

public static void downloadImageImproved() throws FileNotFoundException, IOException
{
    HttpURLConnection connection = null;
    URL url = new URL("http://static1.squarespace.com/static/518f5d62e4b075248d6a3f90/"
        + "t/519bef92e4b02745db13b165/1369173914913/e01.jpg");
    connection = (HttpURLConnection) url.openConnection();
    try(FileOutputStream fos = new FileOutputStream("pic.jpg"));
        InputStream is = connection.getInputStream();
    {
        copyImproved(is, fos);
    }
    finally
    {
        if (connection != null) connection.disconnect();
    }
}
```