

# ESOP - Rekursion / Interface / Exceptions

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Wiederholung



## Basisdatentypen

Signed, two-complement integers

- long - 64 bit
- int - 32 bit
- short - 16 bit
- byte - 8 bit

Floating point numbers

- float - 32 bit
- double - 64 bit

Andere

- char - 16-bit Unicode character
- boolean - true / false

## Referenzdatentypen

Everything with „new“

- Arrays
- Objekte

# Wrapper-Klassen

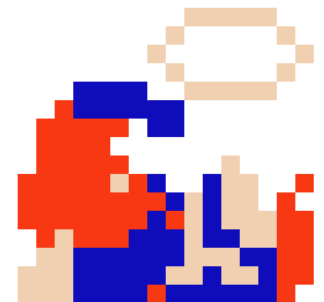


- Byte, Short, Integer, Long, Float, Double
  - verpacken Basisdatentypen
- Wrapper sind Referenzdatentypen
  - keine Basisdatentypen mehr!
- Verpackung größtenteils automatisch
  - Autoboxing & Unboxing
- Siehe auch Boolean

# Fehlerbehandlung & Qualitätssicherung



- Ausnahmebehandlung
  - für den Fall, dass etwas schief geht
- Zusicherungen
  - ein Mechanismus, um strukturiert über die Korrektheit eines Programms nachzudenken und Fehler frühzeitig zu entdecken



# Klassische Muster bei Fehlererkennung



- Fehlerbehandlung überfrachtet den Code
- Algorithmus vs. Fehlerbehandlung?
- Immer tiefer in Vorbedingungen.

```
Aktion1;  
if (Probleme_sind_aufgetreten1)  
    Fehlerbehandlung1;  
else {  
    Aktion2;  
    if (Probleme_sind_aufgetreten2)  
        Fehlerbehandlung2;  
    else {  
        Aktion3;  
        if (Probleme_sind_aufgetreten3)  
            Fehlerbehandlung3;  
        else { ....
```

# Klassische Muster bei Fehlerbehandlung



- Fehlerbehandlung wird nach oben gereicht.
- Aufrufer hat Verantwortung

Vgl. `System.exit(int status)`

- `status != 0` => abnormal termination
- Status wird vom Aufrufer ausgewertet

```
int methode3 () { // Ergebnis = Fehlercode
    Aktion;
    if (Probleme_sind_aufgetreten)
        return 1;
    return 0;
}

int methode2 () { // Ergebnis = Fehlercode
    Aktion;
    int fehler = methode3();
    if (fehler > 0) return 1;
    return 0;
}

void methode1 () {
    Aktion;
    int fehler = methode2();
    if (fehler > 0)
        System.err.println("...nicht hingehauen");
    else ...;
}
```

# Exceptions



Java erlaubt die Trennung von normalem Algorithmus und der Behandlung von Ausnahmesituationen (Exceptions):

- Programm wird einfacher zu verstehen.
- Convenience: Fehlerbehandlung wird einfacher
- Die Fehlerbehandlung wird aufgeschoben
  - Wird an „sinnvoller Stelle“ abgearbeitet
- Aufrufer haben meist erhöhtes Kontextwissen
  - können daher eher sinnvoll auf Fehler reagieren.

# Exceptions



- Eine Exception erzeugt viele Aufrufe in der Java Virtual Machine
  - Stack Trace, usw.
- Entsprechend: Exceptions sind Ausnahme
  - kein Ersatz für Kontrollflusssteuerung

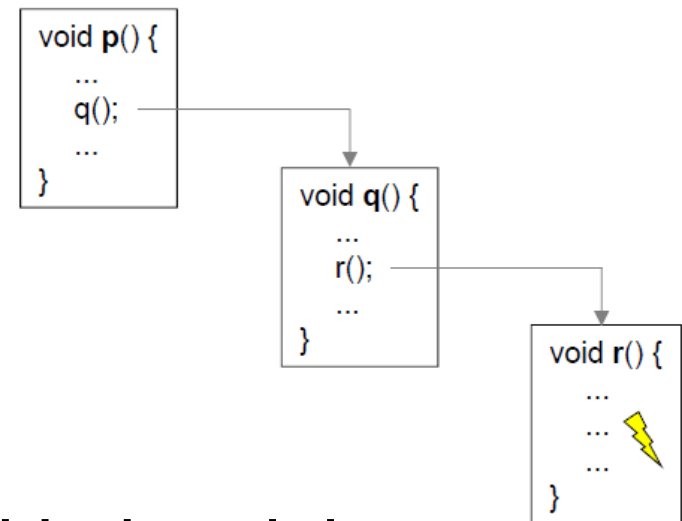


# Optimalfall für Exceptions

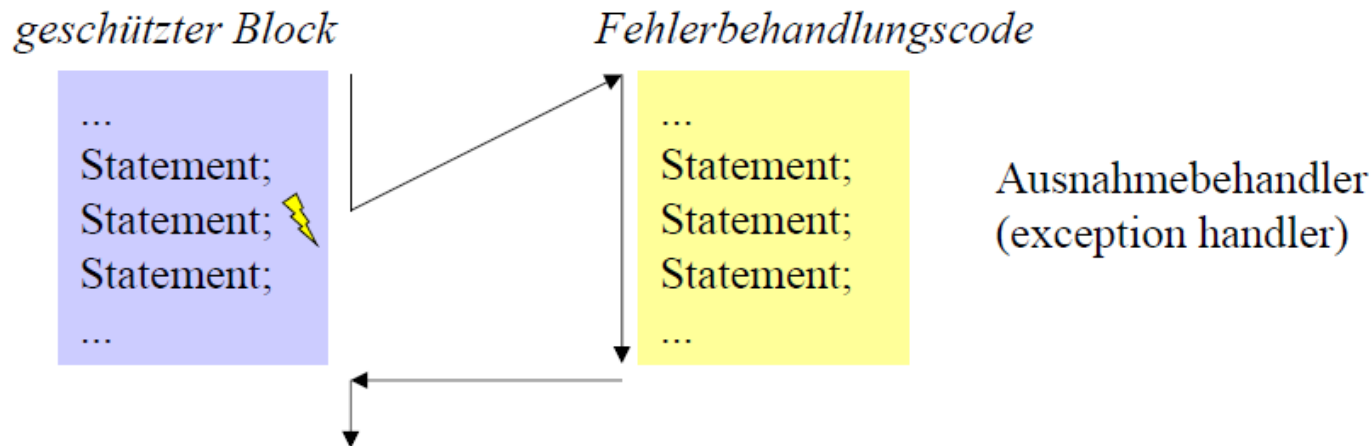


- In `r()` geht etwas schief
  - Was soll passieren?

- Lösung
  - `r()` meldet Fehler an `q()`
  - `q()` meldet an `p()`
  - ... solange bis ihn jemand behandelt.



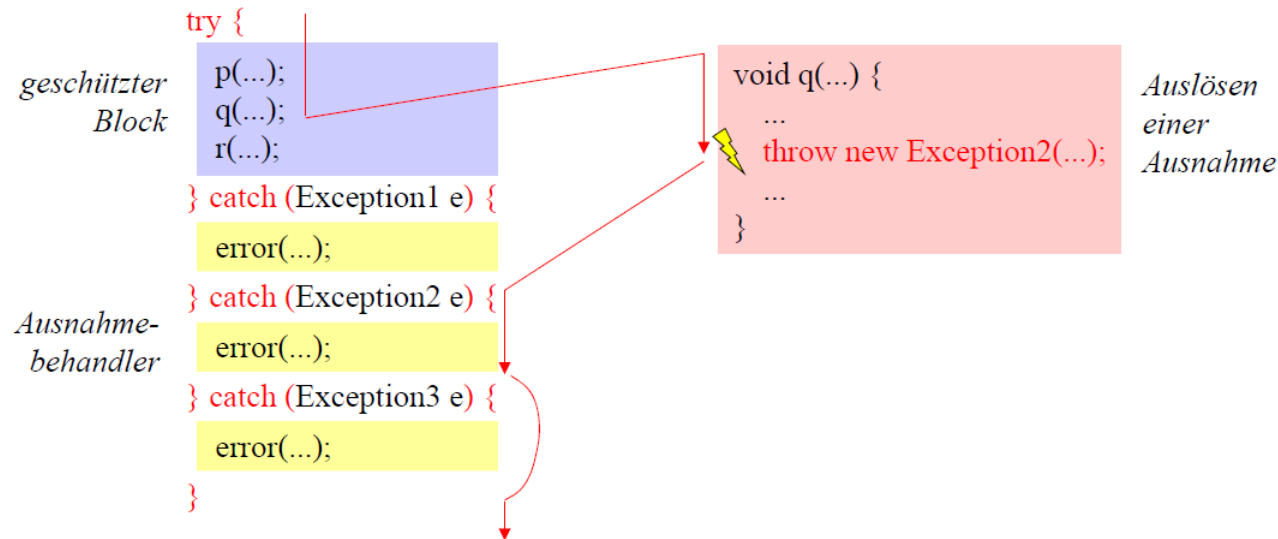
# Fehlerbehandlung in Java



Wenn im geschützten Block ein Fehler (eine Ausnahme) auftritt:

- Ausführung des geschützten Blocks wird abgebrochen
- Fehlerbehandlungscode wird ausgeführt
- Programm setzt nach dem geschützten Block fort

# Try-Anweisungen in Java



- Fehlerfreier Fall und Fehlerfälle sind sauberer getrennt
- Man kann nicht vergessen, einen Fehler zu behandeln
  - Compiler prüft, ob es zu jeder möglichen Ausnahme einen Behandler gibt

# Arten von Ausnahmen



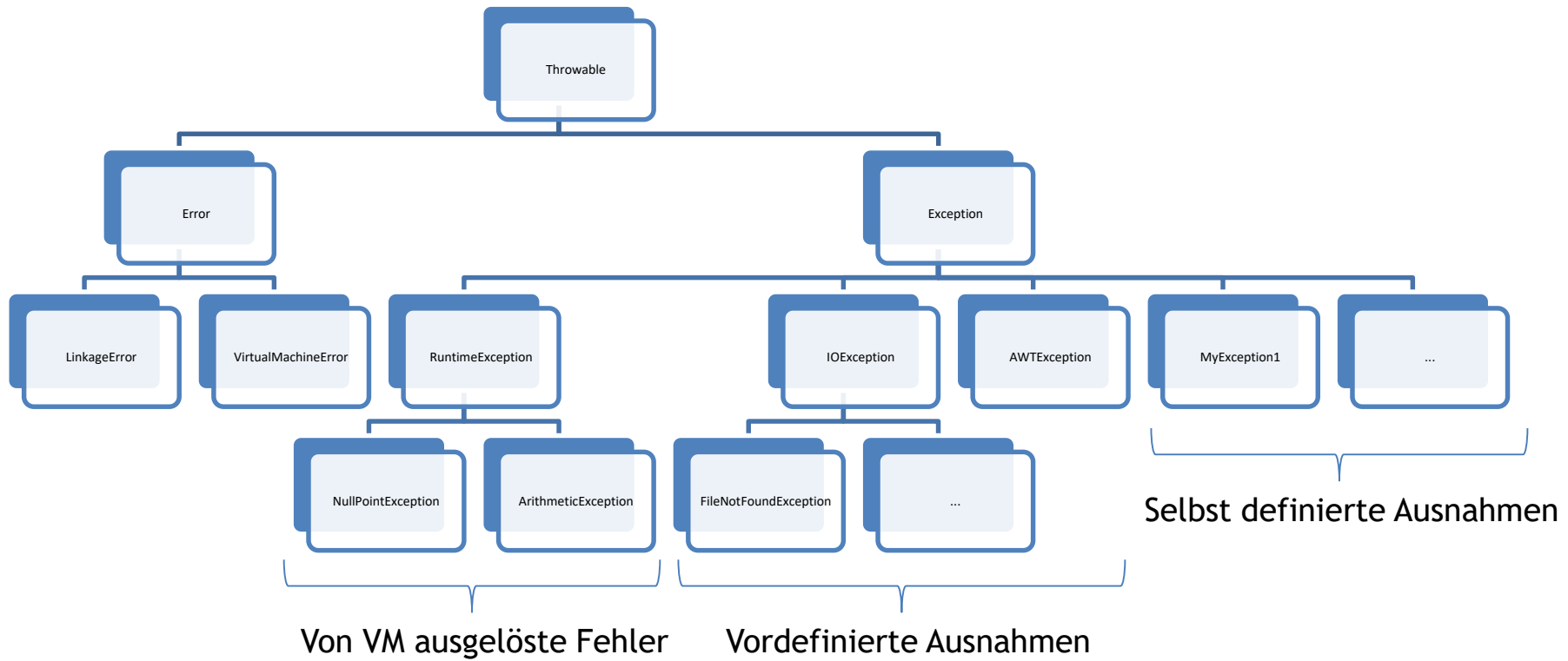
- Laufzeitfehler (Runtime Exceptions) werden von der Java-VM ausgelöst
  - Division durch 0 *ArithmeticException*
  - Zugriff über null-Zeiger *NullPointerException*
  - Indexüberschreitung *ArrayIndexOutOfBoundsException*
- Müssen nicht behandelt werden
- Bei Nichtbehandlung stürzt Programm ab
  - mit einer Fehlermeldung ...

# Arten von Ausnahmen



- Geprüfte Ausnahmen (Checked Exceptions) werden vom Benutzercode ausgelöst (throw-Anweisung)
  - vordefinierte Ausnahmen z.B. `FileNotFoundException`
  - selbst definierte Ausnahmen z.B. `MyException`
- Müssen behandelt werden.
- Compiler prüft, ob sie abgefangen werden.

# Hierarchie der Ausnahmeklassen



# Ausnahmen sind als Klassen umgesetzt.



Fehlerinformationen stehen in einem Ausnahme-Objekt

```
class Exception extends Throwable {  
    Exception(String msg) {...} // erzeugt neues Ausnahmeobjekt mit Fehlermeldung  
    String getMessage() {...} // liefert gespeicherte Fehlermeldung  
    String toString() {...} // liefert Art der Ausnahme und gespeichert Fehlermeldung  
    void printStackTrace() {...} // gibt Methodenaufruflkette aus  
    ...  
}
```

Eigene Ausnameklasse (speichert Informationen über speziellen Fehler)

```
class MyException extends Exception {  
    private int errorCode;  
    MyException(String msg, int errorCode) { super(msg); this.errorCode = errorCode; }  
    int getErrorCode() {...}  
    // toString(), printStackTrace(), ... von Exception geerbt  
}
```

# Throw-Anweisung



- Löst eine Ausnahme aus

```
throw new MyException("invalid operation", 42);
```

- "Wirft" ein Ausnahmeobjekt mit entsprechenden Fehlerinformationen
  - bricht normale Programmausführung ab
  - sucht passenden Ausnahmebehandler (catch-Block)
  - führt Ausnahmebehandler aus und übergibt ihm Ausnahmeobjekt als Parameter
  - setzt nach try-Anweisung fort, zu der der catch-Block gehört



# catch-Blöcke und finally-Block




```
try {  
    ...  
} catch (MyException e) {  
    Out.println(e.getMessage() + ", error code = ", + e.getErrorCode());  
} catch (NullPointerException e) {  
    ...  
} catch (Exception e) {  
    ...  
} finally {  
    ...  
}
```

- Passender catch-Block wird an Hand des Ausnahme-Typs ausgewählt
- catch-Blöcke werden sequentiell abgesucht
- Achtung: speziellere Ausnahme-Typen müssen vor allgemeineren stehen
- Am Ende wird (optionaler) finally-Block ausgeführt
- egal, ob im geschützten Block ein Fehler auftrat oder nicht


# Beispiel finally-Block



```
try {  
    In.open("myfile.txt");  
    ...  
    ...   
    ...  
    In.close();  
} catch (...) {  
    ...  
}
```

**falsch**

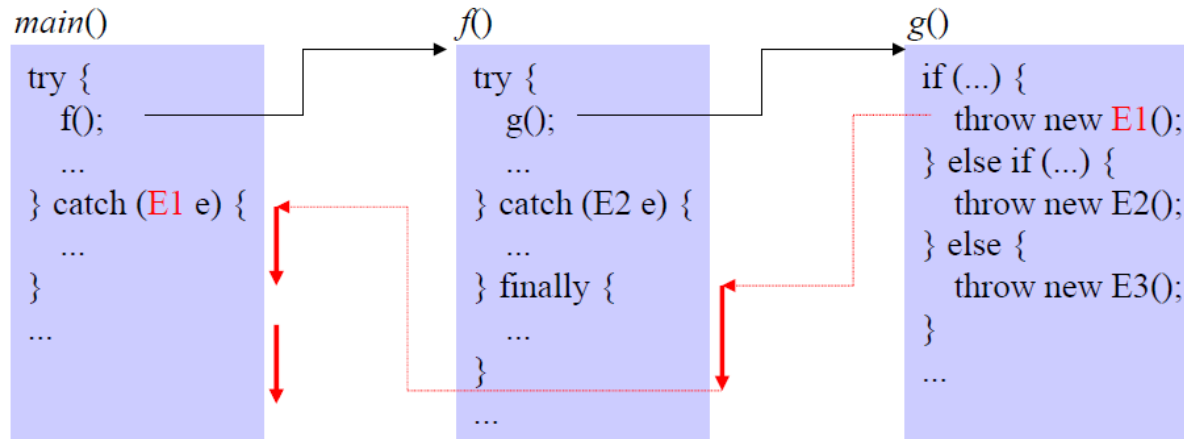
Datei wird im Fehlerfall nicht geschlossen

```
try {  
    In.open("myfile.txt");  
    ...  
    ...   
    ...  
} catch (...) {  
    ...  
} finally {  
    In.close();  
}
```

**richtig**

Datei wird auf jeden Fall geschlossen

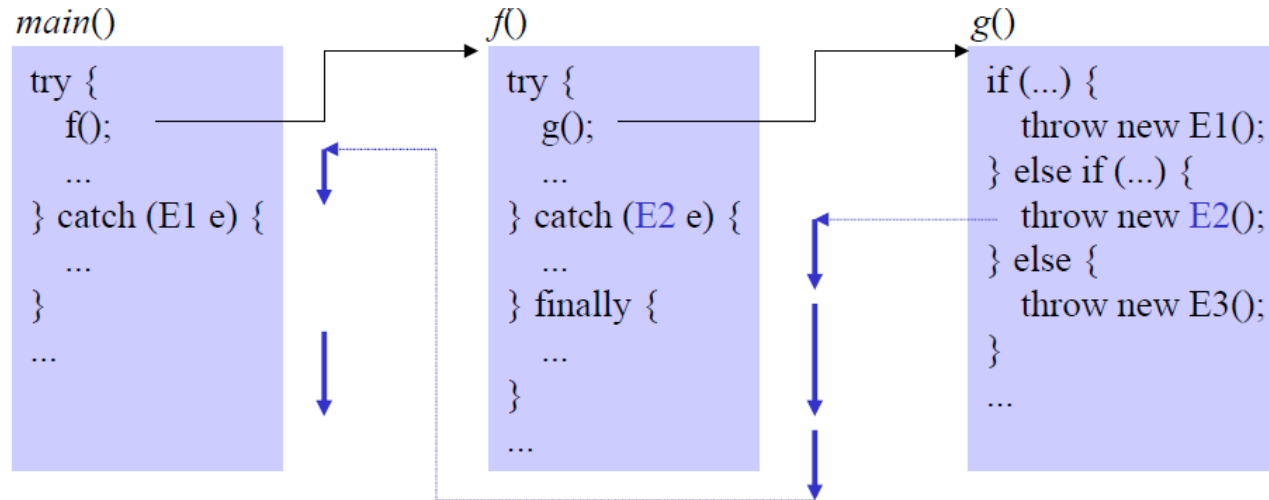
# Ablauflogik bei Ausnahmen



`throw new E1();`

- keine try-Anweisung in `g()` => bricht `g()` ab
- kein passender catch-Block in `f()` => führt finally-Block in `f()` aus und bricht `f()` dann ab
- führt catch-Block für `E1` in `main()` aus
- setzt nach try-Anweisung in `main()` fort

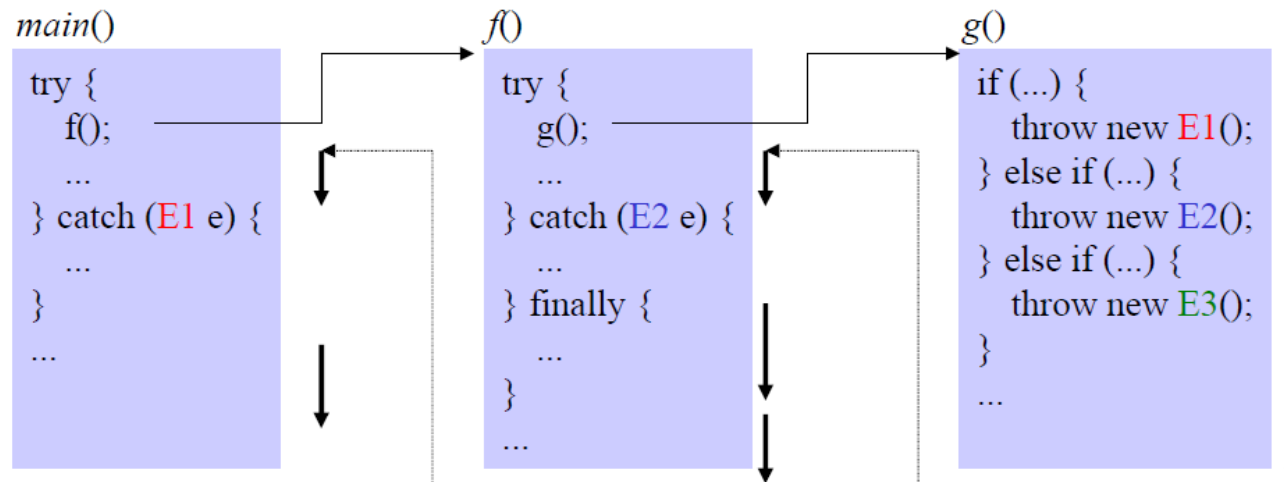
# Ablauflogik bei Ausnahmen



`throw new E2();`

- keine try-Anweisung in *g()* => bricht *g()* ab
- führt catch-Block für E2 in *f()* aus
- führt finally-Block in *f()* aus
- setzt nach try-Anweisung in *f()* fort

# Ablauflogik bei Ausnahmen



`throw new E3();`

- Compiler meldet einen Fehler, weil E3 nirgendwo in der Ruferkette abgefangen wird
- fehlerfreier Fall
  - führt g() zu Ende aus
  - führt try-Block in f() zu Ende aus
  - führt finally-Block in f() aus
  - setzt nach finally-Block in f() fort

# Spezifikation von Ausnahmen im Methodenkopf



Wenn eine Methode eine Ausnahme an den Rufer weiterleitet, muss sie das in ihrem Methodenkopf mit einer *throws-Klausel* spezifizieren

# Spezifikation von Ausnahmen im Methodenkopf



```
void f() {  
    try {  
        ...  
        g();  
        ...  
    } catch (E2 e) {  
        ...  
    }  
}
```

```
void g() throws E2 {  
    try {  
        ...  
        throw new E1();  
        ...  
        throw new E2();  
        ...  
    } catch (E1 e) {  
        ...  
    }  
}
```

# Spezifikation von Ausnahmen im Methodenkopf



- Compiler weiß dadurch, dass `g()` eine E2-Ausnahme auslösen kann.
- Wer `g()` aufruft, muss daher
  - entweder E2 abfangen
  - oder E2 im eigenen Methodenkopf mit einer *throws-Klausel* spezifizieren
- **Man kann nicht vergessen, eine Ausnahme zu behandeln!**



# Daher ...



*void main()* throws E3

```
try {  
    f();  
    ...  
} catch (E1 e) {  
    ...  
}  
...
```

*void f()* throws E1, E3

```
try {  
    g();  
    ...  
} catch (E2 e) {  
    ...  
} finally {  
    ...  
}  
...
```

*void g()* throws E1, E2, E3

```
if (...) {  
    throw new E1();  
} else if (...) {  
    throw new E2();  
} else if (...) {  
    throw new E3();  
}  
...
```

# Rekursion



- Eine Methode  $m()$  heißt *rekursiv*, wenn sie sich selbst aufruft
  - $m() \rightarrow m() \rightarrow m()$  direkt rekursiv
  - $m() \rightarrow o() \rightarrow m()$  indirekt rekursiv

# Rekursion: Fakultät n!



- Definition Fakultät

- $n! = (n-1)! * n$

- $1! = 1$

- Beispiel

- $4! = 4*3! = 4*3*2! = 4*3*2*1! = 4*3*2*1$

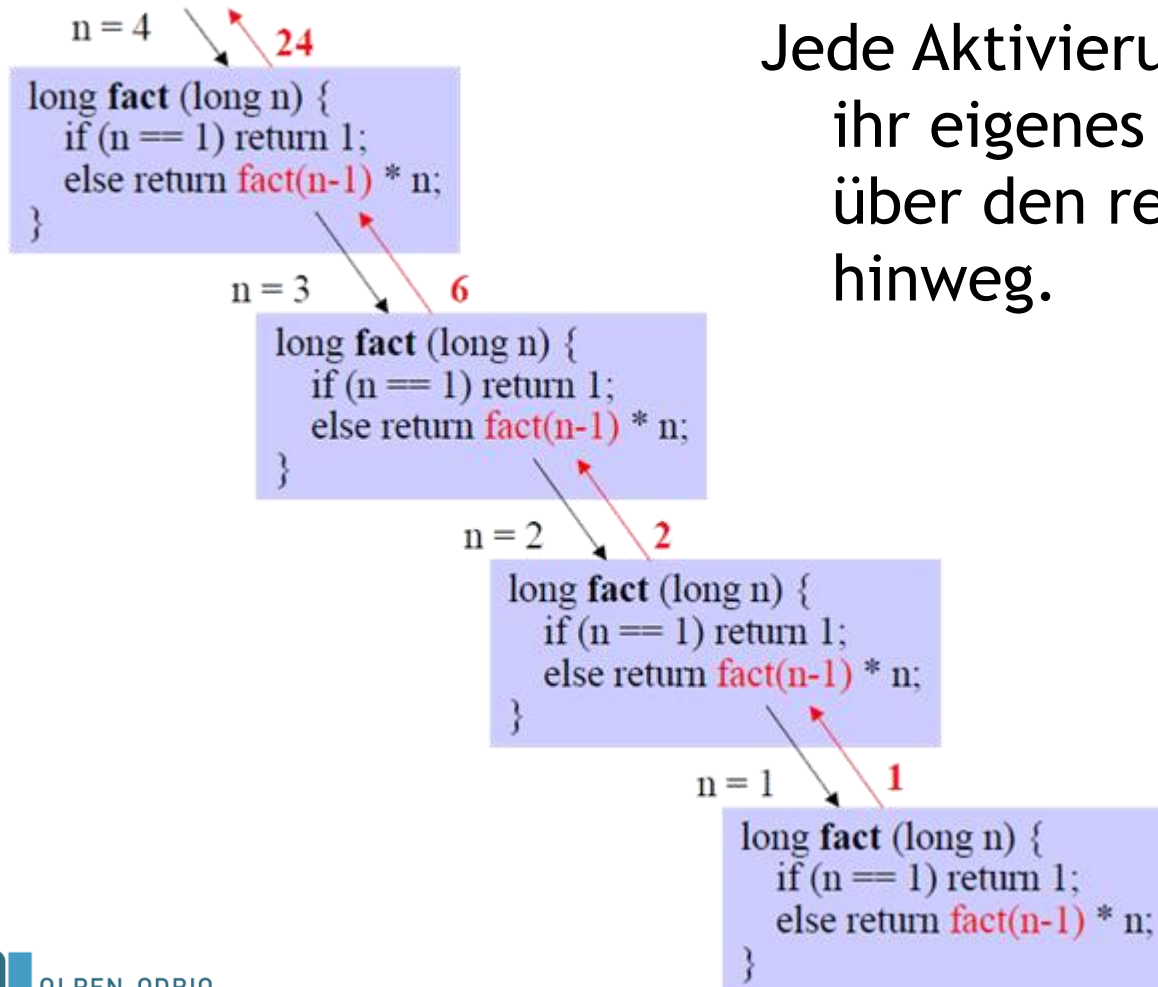
# Rekursion: Fakultät n!



```
long fact (long n) {  
    if (n == 1)  
        return 1;  
    else  
        return fact(n-1) * n;  
}
```

Ende der Rekursion  
bei Erreichen von  
1!

# Ablauf einer rekursiven Methode



Jede Aktivierung von `fact` hat ihr eigenes `n` und rettet es über den rekursiven Aufruf hinweg.

# Beispiel: Binäre Suche Rekursiv



z.B. Suche von 17 (Array muss sortiert sein)

	0	1	2	3	4	5	6	7
a	2	3	5	7	11	13	17	19
	↑			↑			↑	
	low			m			high	

- Index  $m$  des mittleren Element bestimmen
- $17 > a[m] \Rightarrow$  in rechter Hälfte weitersuchen

	0	1	2	3	4	5	6	7
a	2	3	5	7	11	13	17	19
					↑	↑	↑	
					low	m	high	

```
static int search (int elem, int[] a, int low, int high) {  
    if (low > high) return -1; // empty  
    int m = (low + high) / 2;  
    if (elem == a[m]) return m;  
    if (elem < a[m]) return search(elem, a, low, m-1);  
    return search(elem, a, m+1, high);  
}
```

nichtrekursiver Zweig

rekursiver Zweig

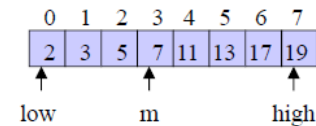
# Beispiel: Binäre Suche Rekursiv



elem = 17, low = 0, high = 7 ↑ 6

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (elem == a[m]) return m;
    if (elem < a[m]) return search(elem, a, low, m-1);
    return search(elem, a, m+1, high);
}
```

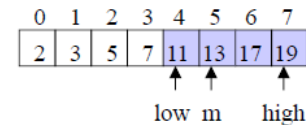
m = 3



low = 4, high = 7 ↓ ↑ 6

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (elem == a[m]) return m;
    if (elem < a[m]) return search(elem, a, low, m-1);
    return search(elem, a, m+1, high);
}
```

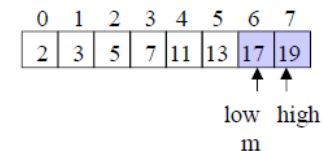
m = 5



low = 6, high = 7 ↓ ↑ 6

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (elem == a[m]) return m;
    if (elem < a[m]) return search(elem, a, low, m-1);
    return search(elem, a, m+1, high);
}
```

m = 6

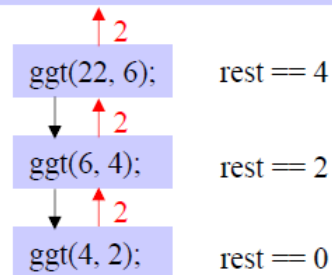


# Beispiel: GGT



*rekursiv*

```
static int ggt (int x, int y) {  
    int rest = x % y;  
    if (rest == 0) return y;  
    else return ggt(y, rest);  
}
```



*iterativ*

```
static int ggt (int x, int y) {  
    int rest = x % y;  
    while (rest != 0){  
        x = y; y = rest;  
        rest = x % y;  
    }  
    return y;  
}
```

Jeder rekursive Algorithmus kann auch iterativ programmiert werden

- rekursiv: meist kürzerer Quellcode
- iterativ: meist kürzere Laufzeit

Rekursion v.a. bei rekursiven Datenstrukturen nützlich (Bäume, Graphen, ...)



# Beispiel: Fibonacci Zahlen



- $F_n = F_{n-1} + F_{n-2}$

```
public static int get(int number) {  
    if (number <= 2)  
        return 1;  
    return get(number-1) + get(number-2);  
}
```

# Interfaces



- Klassenähnlicher Mechanismus
  - zur reinen Verhaltensspezifikation.
- Erlaubt die Definition eines benutzerdefinierten Datentyps von seiner Realisierung zu trennen
  - abstrakter Datentyp.

# Interfaces



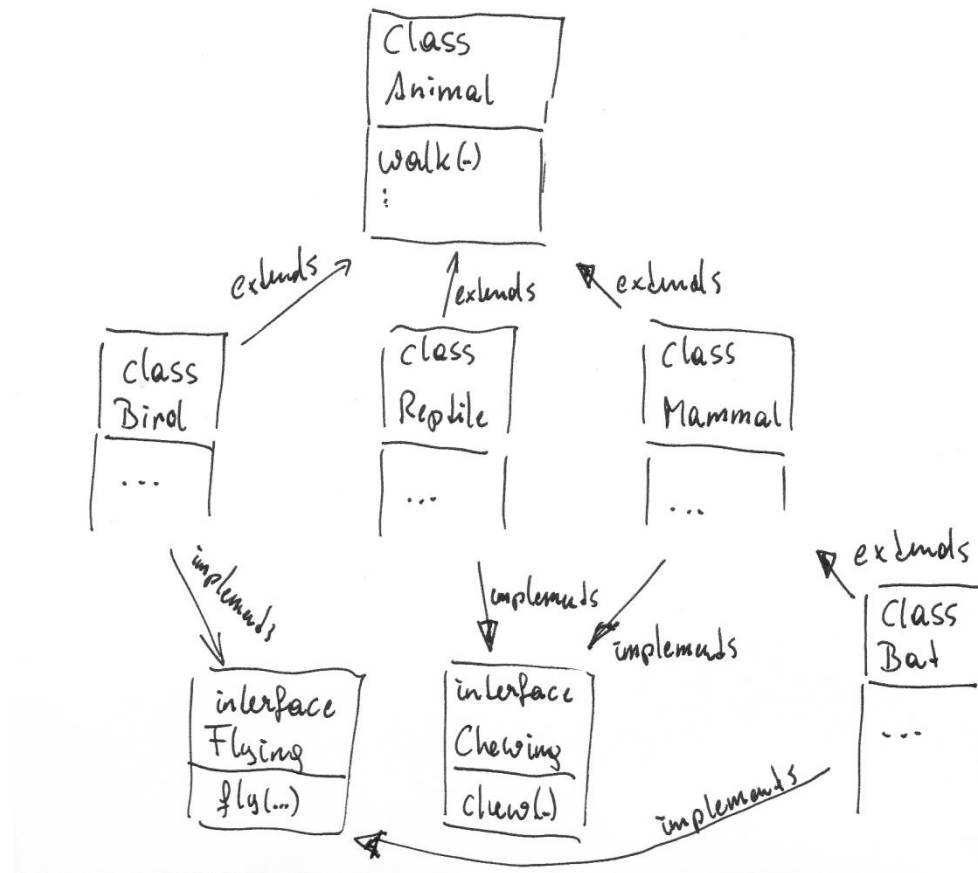
- Spezifikation via interface
- Methoden-Spezifikationen
  - beschreiben, auf welche Nachrichten ein Objekt reagiert
  - ohne Rumpf, also ohne Implementierung.
- Keine Instanzvariablen
  - Aber evt. Konstante

# Interfaces



- Der interface-Name ist in Java als Datentyp verwendbar
- Implementierung via `class`
- Vollständige Methoden
- Instanzvariablen

# Interface Beispiel I



# Interface Beispiel II



[Overview](#) [Package](#) **[Class](#)** [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

Java™ 2 Platform  
Standard Ed. 5.0

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

java.lang

## Interface Iterable<T>

All Known Subinterfaces:

[BeanContext](#), [BeanContextServices](#), [BlockingQueue](#)<E>, [Collection](#)<E>, [List](#)<E>, [Queue](#)<E>, [Set](#)<E>, [SortedSet](#)<E>

All Known Implementing Classes:

[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#), [AbstractSet](#), [ArrayBlockingQueue](#), [ArrayList](#), [AttributeList](#), [BeanContextServicesSupport](#), [BeanContextSupport](#), [ConcurrentLinkedQueue](#), [CopyOnWriteArrayList](#), [CopyOnWriteArraySet](#), [DelayQueue](#), [EnumSet](#), [HashSet](#), [JobStateReasons](#), [LinkedBlockingQueue](#), [LinkedHashSet](#), [LinkedList](#), [PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [SynchronousQueue](#), [TreeSet](#), [Vector](#)

public interface Iterable<T>

Implementing this interface allows an object to be the target of the "foreach" statement.

### Method Summary

<a href="#">Iterator</a> <I>	<a href="#">iterator()</a> Returns an iterator over a set of elements of type T.
------------------------------	---

### Method Detail

**iterator**

[Iterator](#)<I> [iterator\(\)](#)

Returns an iterator over a set of elements of type T.

# Verwendung Interfaces?



- Freigabe minimaler Funktionalität eines abstrakten Datentyps
- Mehrfachvererbung
  - Graph, nicht Baum

# Interface-Beispiele



- Java Interfaces Iterable, Comparable und Serializable



# Wiederholung static



- Zählung der Instanzen als Beispiel
- Singleton & Factory-Pattern