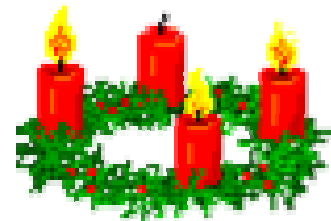


Praktische Informatik 1

Anweisungen, Verzweigungen, Schleifen und Ausnahmen

Thomas Röfer

- Anweisungen
- Verzweigungen
- Schleifen
- Abbruch/Fortsetzung/Rückgabe
- Zusicherung, Ausnahmen
- Übungsblatt



Rückblick „Operatoren und Ausdrücke“

Schreibweisen

| |
|----------|
| Prefix |
| Postfix |
| Infix |
| Roundfix |
| Mixfix |

Arten

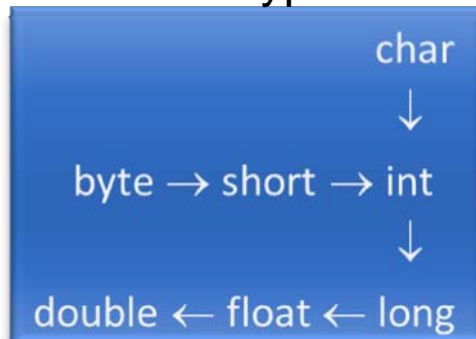
| |
|--------------------------|
| Arithmetische Operatoren |
| Vergleichsoperatoren |
| Logische Operatoren |
| Bitoperatoren |
| Zuweisungsoperatoren |
| Sonstige |

Vorrang/Assoziativität

| | |
|-----------------------|--------|
| .,(),[] | links |
| -,+!,~,++,-- | rechts |
| new,(typ) | rechts |
| *,/,% | links |
| +,- | links |
| <<,>>,>>> | links |
| <,<=,>,>=, instanceof | |
| ==,!= | links |
| & | links |
| ^ | links |
| | links |
| && | links |
| | links |
| ?: | links |
| =,+=,-=,*=,/=,%=,^=, | |
| &=, =,<<=,>>=,>>>= | rechts |

Bindungsstärke

Automatische Typerweiterung

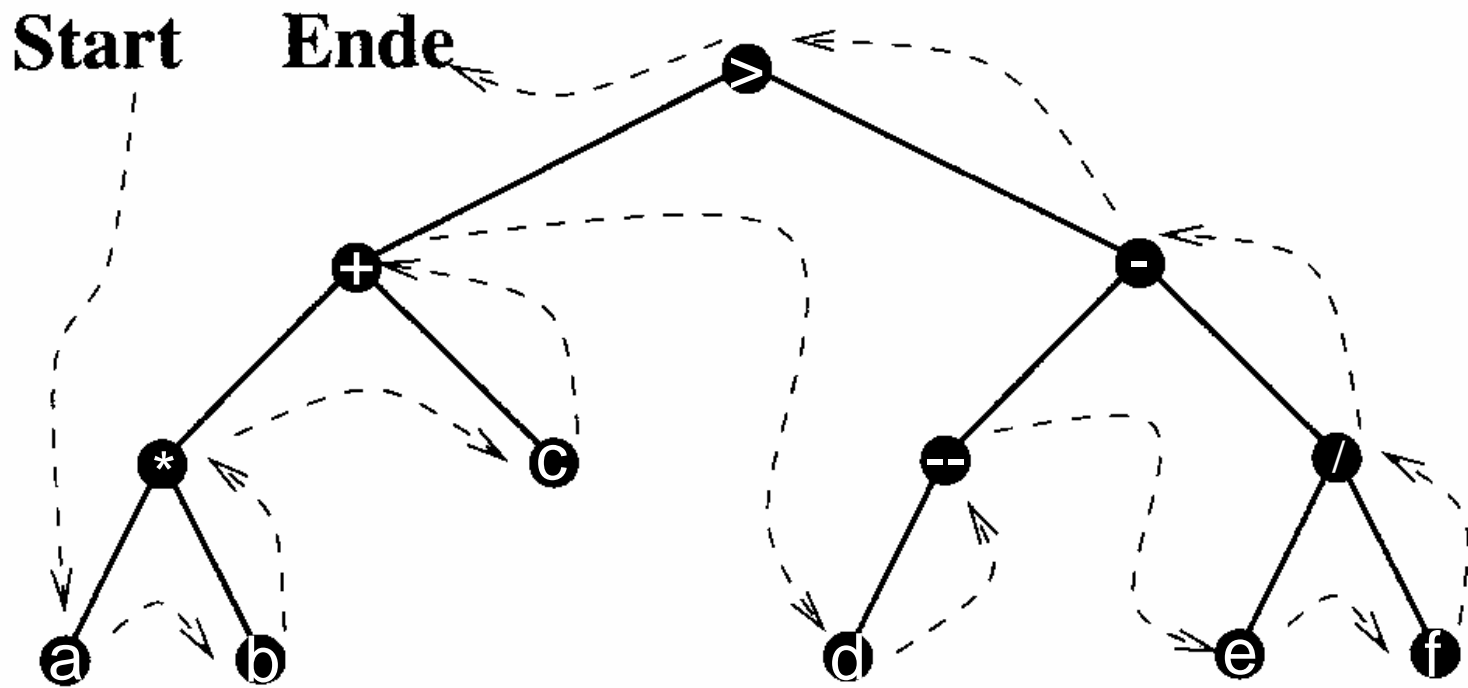


Typanalyse

| | |
|----|--|
| ?: | $\alpha \times \alpha \rightarrow \alpha$ |
| > | $\alpha \times \alpha \rightarrow \text{boolean},$ $\alpha \in \{\text{byte}, \dots, \text{double}\}$ |
| * | $\alpha \times \alpha \rightarrow \alpha,$ $\alpha \in \{\text{int}, \dots, \text{double}\}$ |

Rückblick „Operatoren und Ausdrücke“

- Auswertung von Ausdrücken
 - Erst linker Operand, dann rechter Operand, dann Operator selbst
- $a * b + c > d - - e / f$



Anweisungen

- Leere Anweisung
 - ;
- Variablendeklaration
 - *typ* *bezeichner* [= *ausdruck*] { , *bezeichner* [= *ausdruck*] } ; int
- Zuweisungen
 - *l-wert* [*operator*] = *ausdruck* ;
- Erhöhen/verringern einer Variablen
 - ++ *bezeichner*
 - -- *bezeichner*
 - *bezeichner* ++
 - *bezeichner* --
- Funktionsaufruf
 - [*ausdruck* .] *bezeichner* ([*ausdruck* { , *ausdruck* }]) ;
- Zusicherung
 - **assert** *bedingung* [: *ausdruck*] ;

```
z::B:::  
,,,,,
```

```
i = j * 2;
```

```
++i;
```

```
--i;
```

```
i++;
```

```
i--;
```

```
Xmas.pause(
```

```
assert i
```


Kontrollanweisungen

- Verzweigungen
 - **if** (*bedingung*) *anweisung* [**else** *anweisung*]
 - **switch** (*ausdruck*) { { (**case** *konstausdruck* | **default**)
:
{ *anweisung* } } }
- Schleifen
 - **while** (*bedingung*) *anweisung*
 - **do** *anweisung* **while** (*bedingung*) ;
 - **for** ([*initialisierungen*] ; [*bedingung*] ; [*aktualisierungen*]) *anweisung*
- Ausnahmen fangen
 - **try** { { *anweisung* } }
{ **catch** (*typ bezeichner*) { { *anweisung* } } }
[**finally** { { *anweisung* } }]

Kontrollanweisungen

- Marke (ist keine Anweisung)
 - *bezeichner* :
- Abbruch, Fortsetzung, Rückgabe, Ausnahme
 - **break** [*bezeichner*] ;
 - **continue** [*bezeichner*] ;
 - **return** [*ausdruck*] ;
 - **throw** *ausdruck* ;
- Block
 - { { *anweisung* } }

Verzweigung

- **if (*bedingung*) anweisung [else anweisung]**
- Wenn die Bedingung wahr ist, wird die Anweisung (oder der Anweisungsblock) hinter der *if*-Bedingung ausgeführt
- Falls sie falsch ist und ein *else*-Zweig angegeben wurde, wird dieser ausgeführt
- Anmerkung
 - Ein *else*-Zweig gehört immer zum zuletzt geöffneten *if*

Verzweigung – Beispiele

```
java.io.PrintStream p = System.out;  
if (a == 1) {  
    p.println("a ist eins");  
}
```

```
if (a == 1) {  
    p.println("a ist eins");  
} else {  
    p.println("a ist nicht eins");  
}
```

```
if (a == 1)  
    if (b == 1)  
        p.println("a und b sind eins");  
else  
    p.println("a ist nicht eins");
```



```
if (a == 1) {  
    p.println("a ist eins");  
} else if (a == 2) {  
    p.println("a ist zwei");  
} else if (a == 3 || a == 4) {  
    p.println("a ist drei oder vier");  
} else {  
    p.println("a ist etwas anderes");  
}
```

```
if (a == 1) {  
    if (b == 1)  
        p.println("a und b sind eins");  
} else  
    p.println("a ist nicht eins");
```

Mehrfachverzweigung

- **switch** (*ausdruck*) { { (**case** *konstausdruck* | **default**) : { *anweisung* } } }
- Der hinter *switch* stehende Ausdruck wird ausgewertet
- Dann wird zu der *case*-Marke gesprungen, die den gleichen Wert wie der Ausdruck hat
- Falls hinter keinem *case* ein passender Wert angegeben wurde und eine *default*-Marke vorhanden ist, wird zu dieser gesprungen
- Anmerkungen
 - Der Ausdruck muss vom Typ *int* (oder automatisch erweiterbar) oder ein Aufzählungstyp (*enum*) sein
 - Hinter *case* können nur Übersetzungszeitkonstanten von kompatibellem Typ stehen
 - Es darf maximal eine *default*-Marke angegeben werden
 - Die Ausführung wird auch über Marken hinweg fortgesetzt, entweder bis zum Ende der *switch*-Anweisung oder bis zur Anweisung *break*

Mehrfachverzweigung – Beispiel 1

```
java.io.PrintStream p = System.out;
if (a == 1) {
    p.println("a ist eins");
} else if (a == 2) {
    p.println("a ist zwei");
} else if (a == 3 || a == 4) {
    p.println("a ist drei oder vier");
} else {
    p.println("a ist etwas anderes");
}
```

In *switch*-Anweisungen müssen die einzelnen *case*-Blöcke fast immer mit einem *break* abgeschlossen werden!

```
java.io.PrintStream p = System.out;
switch (a) {
    case 1:
        p.println("a ist eins");
        break;
    case 2:
        p.println("a ist zwei");
        break;
    case 3:
    case 4:
        p.println("a ist drei oder vier");
        break;
    default:
        p.println("a ist etwas anderes");
}
```


Mehrfachverzweigung – Beispiel 2

```
class Calendar {  
    enum Month {JANUARY, FEBRUARY,  
                MARCH, APRIL, MAY, JUNE, JULY,  
                AUGUST, SEPTEMBER, OCTOBER,  
                NOVEMBER, DECEMBER}  
    static int days (Month month, int year) {  
        switch (month) {  
            case JANUARY:  
            case MARCH:  
            case MAY:  
            case JULY:  
            case AUGUST:  
            case OCTOBER:  
            case DECEMBER:  
                return 31;
```

```
            case APRIL:  
            case JUNE:  
            case SEPTEMBER:  
            case NOVEMBER:  
                return 30;  
            case FEBRUARY:  
                return year % 4 == 0 &&  
                    year % 100 != 0 ||  
                    year % 400 == 0 ? 29 : 28;  
            default:  
                return 0;  
        }  
    }  
}
```

Abweisende Schleife

- **while** (*bedingung*) *anweisung*
- Wenn die Bedingung wahr ist, wird die Anweisung (bzw. der Anweisungsblock) ausgeführt
- Dies wird solange wiederholt, bis die Bedingung nicht mehr wahr ist
- Anmerkungen
 - Falls die Bedingung bereits zu Anfang falsch ist, wird die Anweisung (bzw. der Anweisungsblock) niemals ausgeführt
 - Die Anweisung (bzw. der Anweisungsblock) sollte die in der Bedingung verwendeten Variablen verändern, sonst terminiert die Schleife nicht (außer durch *break*, *continue*, *return* oder *throw*)

```
while (i > 0) {  
    System.out.println(100 / i);  
    --i;  
}
```


Annehmende Schleife

- **do anweisung while (bedingung)**
- Die Anweisung (bzw. der Anweisungsblock) wird ausgeführt
- Danach wird die Bedingung getestet.
 - Ist sie wahr, werden die Anweisung (bzw. der Anweisungsblock) und der Bedingungstest wiederholt, solange, bis die Bedingung nicht mehr wahr ist
- Anmerkungen
 - Falls die Bedingung bereits zu Anfang falsch ist, wird die Anweisung (bzw. der Anweisungsblock) dennoch einmal ausgeführt
 - Die Anweisung (bzw. der Anweisungsblock) sollte die in der Bedingung verwendeten Variablen verändern, sonst terminiert die Schleife nicht (außer durch *break*, *continue*, *return* oder *throw*)

```
// Fehler bei i = 0 zu Beginn  
do {  
    System.out.println(100 / i);  
    --i;  
} while (i > 0);
```

Zählschleife

- **for** ([*initialisierungen*] ; [*bedingung*] ; [*aktualisierungen*])
 anweisung
 - *initialisierungen* = [*typ*] *zuweisung* { , *zuweisung* }
 - *aktualisierungen* = *aktualisierung* { , *aktualisierung* }
 - *aktualisierung* = (++ | --) *bezeichner* | *bezeichner* (++ | --) |
 zuweisung
- Zuerst werden die Initialisierungen ausgeführt
 - Werden dabei Variablen deklariert, endet ihre Lebenszeit nach Ende der Schleife
- Ist die Bedingung wahr, werden erst die Anweisung (bzw. der Anweisungsblock) und dann die Aktualisierungen ausgeführt
- Dann wird wieder mit der Überprüfung der Bedingung fortgefahren

$$\sum_{i=0}^{n-1} v_i \triangleq$$

```
int sum = 0;
for (int i = 0; i < n; ++i)
    sum += v[i];
```

Aufzählungsschleife

- **for** (*typ bezeichner : bezeichner*) *anweisung*
- Schleife läuft über alle Elemente einer Reihung von Index 0 bis Index *length* – 1
- Jedes Element wird vor Schleifendurchlauf an Platzhalter zugewiesen
- Veränderungen am Platzhalter wirken sich nicht auf die Elemente der Reihung aus
- Hinweis
 - Funktioniert auch mit Klassen, die die Schnittstelle *Iterable<T>* implementieren

```
int sum(int[] a) {  
    int s = 0;  
    for (int e : a) {  
        s += e;  
    }  
    return s;  
}
```

Zählschleife – Beispiele

```
static void primes(int n) {  
    boolean[] sieve = new boolean[n + 1];  
    for (int i = 2; i <= n; ++i) {  
        if (!sieve[i]) {  
            System.out.println(i);  
            for (int j = 2 * i; j <= n; j += i) {  
                sieve[j] = true;  
            }  
        }  
    }  
}
```

```
for(;;) {  
    // ...  
    if (bedingung)  
        break;  
    //...  
}
```

Zähl-/Aufzählungsschleife – Beispiele

```
static int findFirst(String[] a, String b) {  
    int i;  
    for (i = 0; i < a.length && !a[i].equals(b); ++i);  
    return i < a.length ? i : -1;  
}
```

```
static int findLast(String[] a, String b) {  
    int i;  
    for (i = a.length; i-- > 0 && !a[i].equals(b););  
    return i;  
}
```

```
static int product(int[][] a) {  
    p = 1;  
    for (int[] a0 : a) {  
        for (int a1 : a0) {  
            p *= a1;  
        }  
    }  
    return p;  
}
```

Analogien zwischen Schleifen

```
while (bedingung)  
  anweisung
```



```
if (bedingung)  
  do  
    anweisung  
  while (bedingung);
```

```
do  
  anweisung  
while (bedingung)
```



```
anweisung  
while (bedingung)  
  anweisung
```



```
boolean first = true;  
while (first || (bedingung)) {  
  anweisung  
  first = false;  
}
```


Analogien zwischen Schleifen

```
for (initialisierungen; bedingung; aktualisierungen)  
  anweisung
```



```
{  
  initialisierungen;  
  while (bedingung) {  
    anweisung  
    aktualisierungen; // ',' → ';' ;  
  }  
}
```

```
for (typ element : reihung)  
  anweisung
```

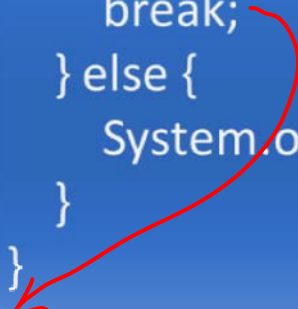


```
for (int i = 0; i < reihung.length; ++i) {  
  typ element = reihung[i];  
  anweisung  
}
```

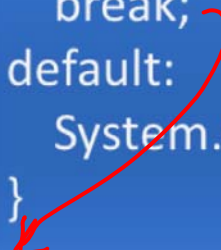
Abbruchanweisung

- **break** [*bezeichner*] ;
- *break* ohne Angabe einer Marke setzt die Ausführung hinter der aktuellen Schleife/switch-Anweisung fort

```
while (true) {  
    if (i == 0) {  
        break;  
    } else {  
        System.out.println(100 / i--);  
    }  
}
```



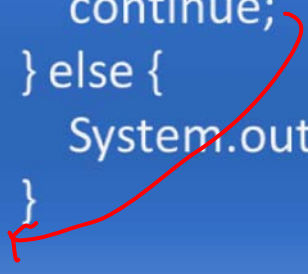
```
switch (i) {  
    case 0:  
        break;  
    default:  
        System.out.println(100 / i--);  
}
```



Fortsetzungsanweisung

- **continue** [*bezeichner*] ;
- *continue* ohne Angabe einer Marke leitet den nächsten Schleifendurchlauf ein

```
for (int i = 10; i >= -10; --i) {  
    if (i == 0) {  
        continue;  
    } else {  
        System.out.println(100 / i);  
    }  
}
```



Abbruch-/Fortsetzung mit Marke

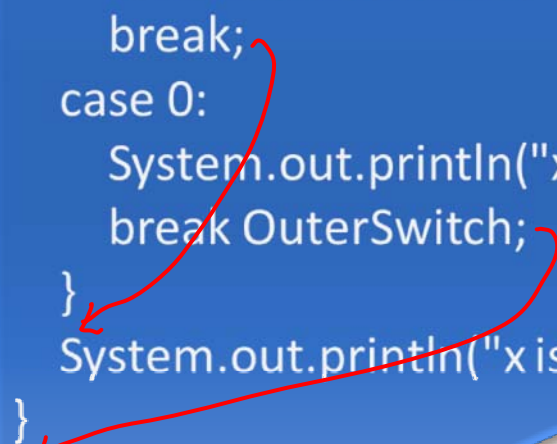
- *bezeichner* :
- Vor nahezu jeder Anweisung kann eine Marke stehen (nur vor Deklarationen nicht)
- *break* mit Angabe einer Marke setzt die Ausführung hinter der markierten Anweisung (Block) fort
- *continue* mit Angabe einer Marke leitet den nächsten Durchlauf der markierten Schleife ein

```
OuterLoop:  
for (int x = 0; x < 10; ++x) {  
    for (int y = 0; y < 10; ++y) {  
        if (a[x][y] == 42) {  
            break;  
        } else if (a[x][y] == 43) {  
            break OuterLoop;  
        }  
    }  
}
```

```
OuterLoop:  
for (int x = 0; x < 10; ++x) {  
    for (int y = 0; y < 10; ++y) {  
        if (a[x][y] == 42) {  
            continue;  
        } else if (a[x][y] == 43) {  
            continue OuterLoop;  
        }  
    }  
}
```

Abbruchanweisung mit Marke

```
OuterSwitch:  
switch (x) {  
case 0:  
    switch (y) {  
case 1:  
    System.out.println("y ist 1");  
    break;  
case 0:  
    System.out.println("x und y sind 0");  
    break OuterSwitch;  
    }  
    System.out.println("x ist 0");  
}
```



Rückgabeanweisung

- **return** [*ausdruck*] ;
- Die *return*-Anweisung beendet die Ausführung der aktuellen Funktion und kehrt zum Aufrufer zurück
- Der Typ des Ausdrucks hinter *return* muss kompatibel zum Rückgabotyp der Funktion sein
- Ist der Rückgabotyp einer Funktion *void*, darf hinter *return* kein Ausdruck stehen
- Ist der Rückgabotyp nicht *void*, muss jeder mögliche Ausführungspfad durch die Funktion in einer *return*-Anweisung enden

```
static int factorial(int n) {  
    if (n > 0) {  
        return n * factorial(n - 1);  
    } else {  
        return 1.0;  
    } // Falscher Rückgabotyp  
}
```

```
static int factorial(int n) {  
    if(n > 0) {  
        return n * factorial(n - 1);  
    }  
    // Fehlendes return  
}
```

Zusicherung

- **assert** *bedingung* [: *ausdruck*]
- *assert* überprüft eine Zusicherung
- Ist diese nicht gegeben, wird eine sog. Ausnahme erzeugt
- Anmerkungen
 - Standardmäßig ignorieren Java-Programme *assert*-Anweisungen, es sei denn, die Option *-ea* wird angegeben
 - In BlueJ kann man Zusicherungen aktivieren, indem man in der *bluej.defs* den folgenden Eintrag erweitert:
bluej.vm.args=-server -Xincgc -ea

```
static int factorial(int n) {  
    assert n >= 0 && n <= 12 :  
        n + "! nicht definiert";  
    if(n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

Ausnahmen (Exceptions)

- **throw** *ausdruck* ;
 - Exceptions dienen zum Mitteilen von Fehlerzuständen
 - An der Stelle, an der die Exception ausgelöst wird, wird die Ausführung des Programms unterbrochen und an anderer Stelle wieder aufgenommen
 - „Geworfene“ Objekte müssen einen Typ haben, der von *Throwable* abgeleitet ist
- *funktionskopf* [**throws** *typ* { , *typ* }] *funktionsrumpf*
 - Durch die Angabe einer *throws*-Klausel hinter dem Funktionskopf werden Aufrufer einer Funktion gezwungen, die genannte Exception zu behandeln
 - Allerdings kann der Aufrufer ebenfalls eine *throws*-Klausel hinter seinem Funktionskopf verwenden, um die Behandlung an seinen Aufrufer weiterzuleiten

Ausnahmen – Beispiel

```
class Buchführung {
    static final int gewinnspanne = 150;
    static final int mwst = 19;
    static double berechneVKPreis(double ekpreis) throws Exception {
        if (ekpreis <= 0) {
            throw new Exception("Negativer oder Gratiseinkaufspreis ist nicht erlaubt");
        } else if (ekpreis * 100 - (int) (ekpreis * 100) > 0) {
            throw new Exception("Mehr als zwei Nachkommastellen sind nicht erlaubt");
        }
        double zwischenpreis = ekpreis * (1 + gewinnspanne / 100.0);
        double ergebnis = zwischenpreis * (1 + mwst / 100.0);
        return (int) (ergebnis * 100) / 100.0;
    }
}
```

Ausnahmen (Exceptions)

- `try { { anweisung } }`
`{ catch (typ bezeichner) { { anweisung } } }`
`[finally { { anweisung } }]`
- Das Behandeln einer Exception bezeichnet man als „fangen“
- Für jeden Typ von Exception kann man einen eigenen *catch*-Block schreiben
- Ein *finally*-Block wird immer ausgeführt, unabhängig davon, ob Exceptions geworfen wurden oder nicht
 - Auch, wenn die Methode im *try*-Block mit *return* beendet wird
- Wird eine Exception nicht gefangen, wird das Programm abgebrochen und eine Fehlermeldung ausgegeben

Ausnahmen – Beispiele

```
static void berechneVKPreise(double[] ekpreise) {  
    for (double ekp : ekpreise) {  
        try {  
            System.out.println("EK-Preis " + ekp + " € -> VK-Preis " +  
                berechneVKPreis(ekp) + " €");  
        } catch (Exception e) {  
            System.out.println(e.getMessage() + ": " + ekp + " €");  
        }  
    }  
}
```

Ausnahmen – Beispiele

```
class Test {  
    static void test1(int n)  
        throws Exception {  
        try {  
            if (n == 0) {  
                throw new Exception("n == 0");  
            }  
            System.out.println("n != 0");  
            return;  
        } finally {  
            System.out.println("Immer!");  
        }  
    }  
}
```

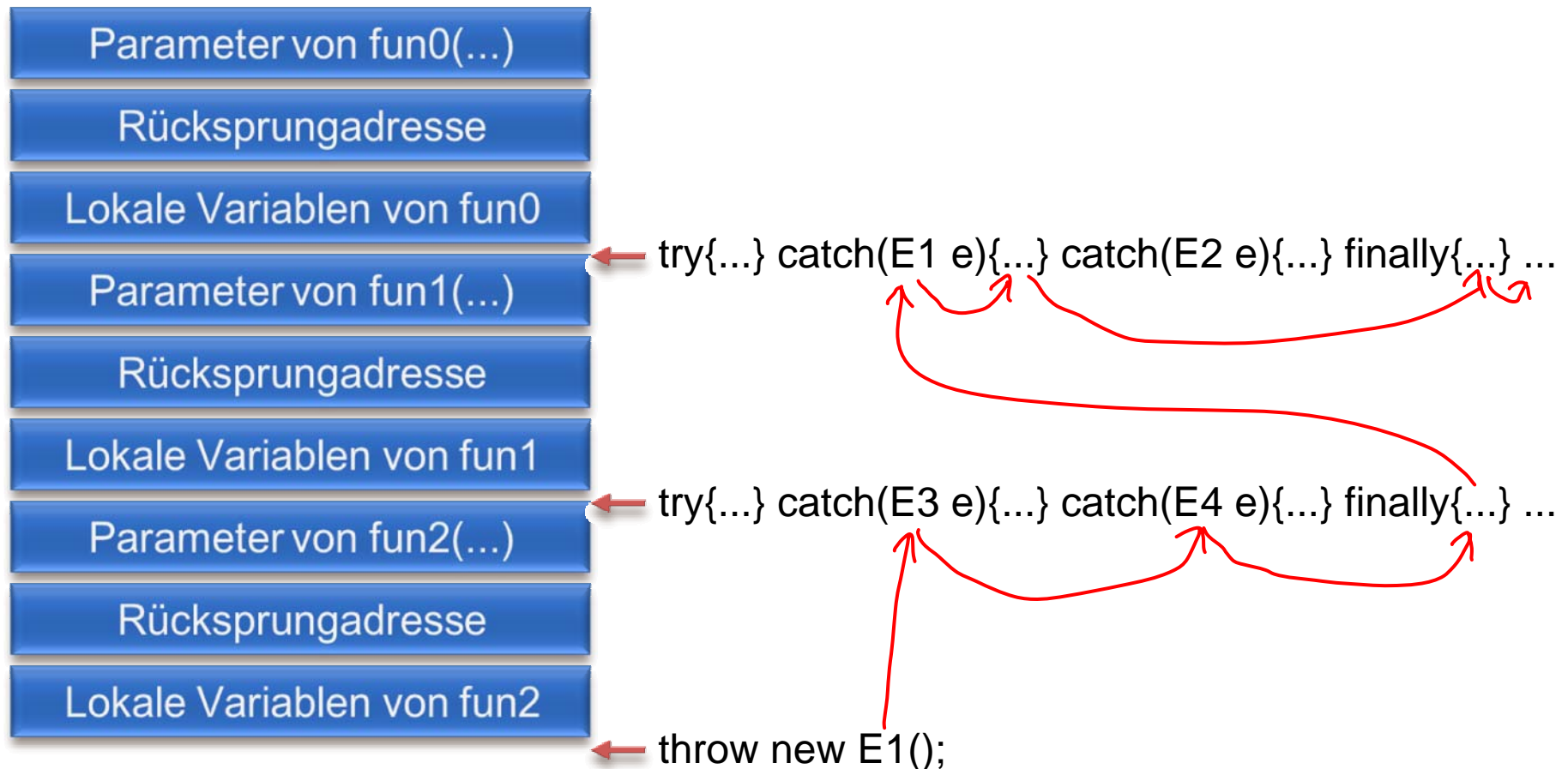
```
static void test2(int n) {  
    try {  
        test1(n);  
    } catch (Exception e) {  
        System.out.println(  
            e.getMessage());  
    }  
}
```

Ausnahmen – Beispiele

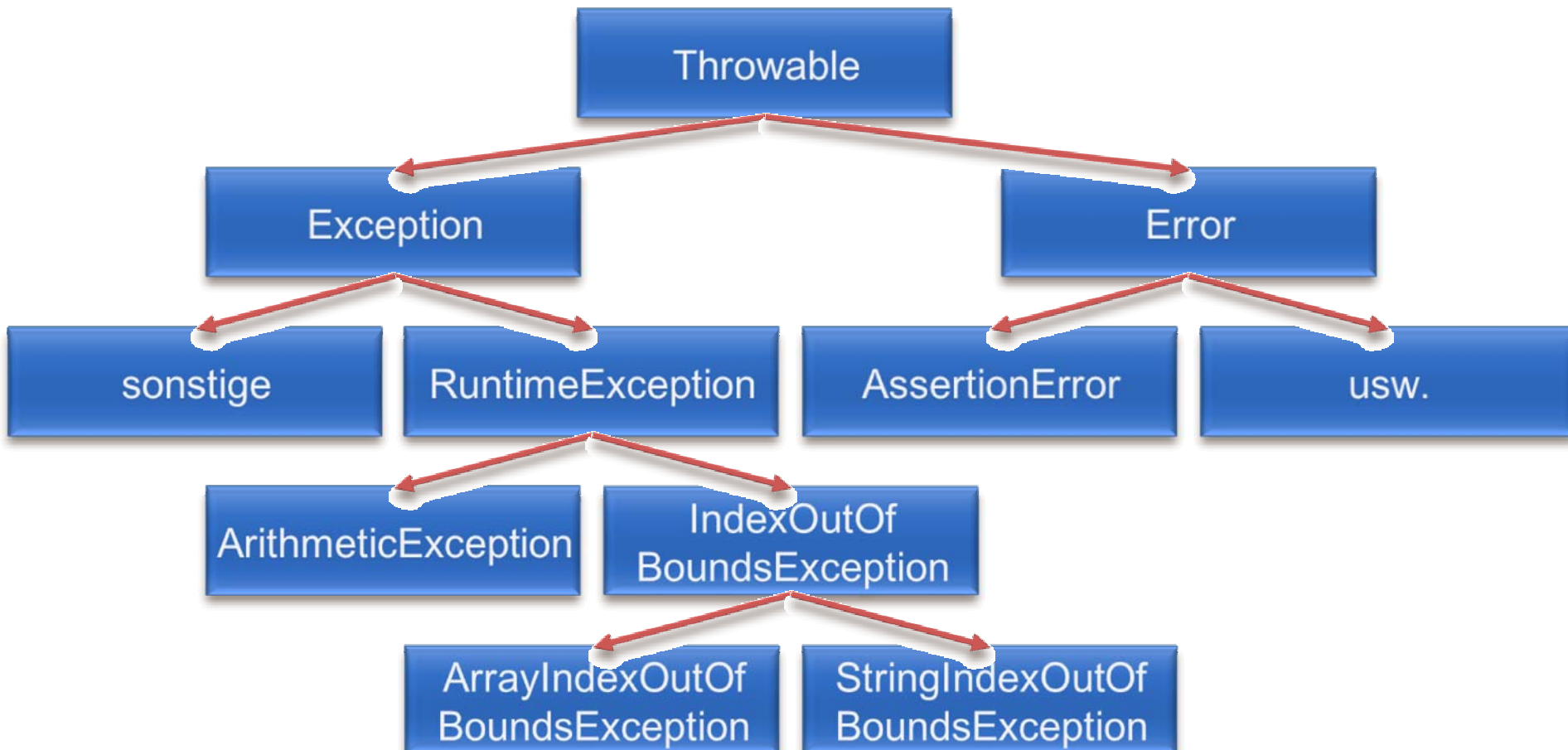
```
class Factorial {  
    static void f(int n, int r)  
        throws Exception {  
        if(n == 0) {  
            throw new Exception("" + r);  
        } else {  
            f(n - 1, r * n);  
        }  
    }  
}
```

```
static int factorial(int n) {  
    try {  
        f(n, 1);  
    } catch (Exception e) {  
        return new Integer(e.getMessage());  
    }  
    return 0; // Wird nie erreicht  
}
```

Der Weg einer Ausnahme



Die Exception-Hierarchie



Übungsblatt 8

- Aufgabe 1
 - *while* durch *if* ersetzen?
 - *if* durch *while* ersetzen?
- Aufgabe 2
 - Anwendung von Bit-Operatoren
 - \ggg , \ll , $\&$, $|$, \sim
 - Umgang mit *long*-Werten
 - $1 \ll 63 \neq 1L \ll 63$
 - Bonusaufgabe!

Praktische Informatik I WS 2007/08

Übungsblatt 8

Abgabe: 09.01.08

Aufgabe 1 Der Grinch hat die Anweisungen geklaut (30%)

Der Grinch mag nicht nur kein Weihnachten, sondern auch kein Java. Daher hat er fast alle Java-Kontrollanweisungen entwendet (*do* ... *while*, *switch* usw., aber auch den *?:*-Operator). Übrig geblieben sind nur *while* und *if-else*. Eine Anweisung von beiden könnt ihr verstecken, um sie seinem diebischen Zugriff zu entziehen und somit auch weiterhin programmieren zu können. Zeigt, dass sich eine der beiden Anweisungen vollständig durch die andere ersetzen lässt. Diskutiert auch, warum die Ersetzung in der anderen Richtung im Allgemeinen nicht möglich ist.

Aufgabe 2 Schwarzweiße Weihnacht (70%)

Damit der Weihnachtsmann die etwa 2 Mrd. Kinder auf unserem Planeten in einer Nacht mit Geschenken beliefern kann, muss er sich ganz schön beeilen. Er bewegt sich so schnell, dass ihn noch nie jemand gesehen hat. Aber mit moderner Informationstechnik können wir ihn sichtbar machen. Die im Archiv *uebung08* bereitgestellte Klasse *Xmas* enthält vier Momentaufnahmen vom Weihnachtsmann und seinem getreuen Rentier Rudolph, sowie ein Bild des typischen Lebensraums der beiden. Die Bilder sollt ihr zum Leben erwecken.

Jedes Bild liegt als Reihung von *longs* vor, wobei jeder Eintrag der Reihung einer Zeile des Bildes entspricht. Jede Zeile besteht aus 64 Pixeln, die in den 64 Bits des *longs* gespeichert sind, wobei ein gesetztes Bit der Farbe Schwarz entspricht und ein gelöschtes Bit der Farbe Weiß. Das LSB repräsentiert das linke Pixel in der Zeile, das MSB das rechte. Die Klasse *Xmas* enthält neun solche Bilder. Eines ist das Hintergrundbild. Vier weitere bilden eine Animation der Bewegung des Weihnachtsmanns. Noch vier weitere stellen eine Maske für die Animation dar, damit man diese vor dem Hintergrund ablaufen lassen kann, ohne dass dieser durchscheint.

a) Schreibt eine Methode, die ein Bild in Form einer Reihung von *longs* als Parameter bekommt, die Konsole löscht und danach das übergebene Bild ausgibt, indem für jedes nicht-gesetzte Bit ein Leerzeichen ausgegeben wird und für jedes gesetzte der Buchstabe M.

b) Schreibt eine Methode, die aus einem Hintergrundbild, einem Vordergrundbild, einem Maskenbild und einer seitlichen Verschiebung $s \in \{-63 \dots 63\}$ ein Bild generiert. Dazu werden zuerst das Vordergrundbild und die Maske um den in s angegebenen Versatz seitlich verschoben. Die Maske wird so mit dem Hintergrundbild verknüpft, dass alle Bits im Hintergrundbild gelöscht werden, die in der Maske gesetzt sind. Dadurch wird der Bereich des Bildes gelöscht, in den der Vordergrund gezeichnet werden soll. In dem so entstandenen Bild werden danach alle Bits gesetzt, die auch im Vordergrundbild gesetzt sind. Das so erzeugte Bild ist das Ergebnis der Methode. Beachtet, dass all diese Operationen zeilenweise und durch die Anwendung von \ggg , \ll , $\&$, $|$ und \sim effizient durchgeführt werden können.

Übungsblatt 8 – Aufgabe 2

- Klasse Xmas

- background 
- foregrounds 
- masks 

