

Einführung in die Informatik II
Ausnahmen

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

Sommersemester 2001

30.Mai / 11. Juni 2001

Überblick

- ❖ Ausnahme: Ein Ereignis, das den normalen Programmfluss ändert
- ❖ Ausnahmen in Java
- ❖ Modellierung von Ausnahmen
- ❖ Programmierung von Ausnahmen
- ❖ Dokumentation von Ausnahmen
- ❖ Behandlung von Vertragsbrüchen mit Ausnahmen

Ziel der Vorlesung

- ❖ Sie verstehen die Konzepte der Ausnahmen und der Ausnahmebehandlung.
- ❖ Sie können Ausnahmen modellieren, als Unterklassen von Java-Ausnahmen implementieren und in Javadoc dokumentieren.
- ❖ Sie verstehen, wie man Ausnahmebehandlung implementiert.
 - ◆ Sie sind in der Lage, den try/catch/finally-Rahmen in Java zu benutzen.
- ❖ Sie verstehen, wie man Verträge des Spezifikationsmodell mit Hilfe von Ausnahmen im Quellprogramm formulieren kann.

Fehlerbehandlung in Software-Systemen

- ❖ Einer der größten Problembereiche beim Entwurf von Software-Systemen ist der Umgang mit möglichen Fehlern.
Beim Entwurf von Software müssen Sie sich immer die folgenden 2 Fragen stellen:
 - ◆ Was kann falsch laufen?
 - ◆ Was mache ich, wenn etwas falsch läuft?
- ❖ Im Kontext von Verträgen:
 - ◆ Was ist, wenn der Vertrag falsch formuliert worden ist, oder wenn es gar keinen Vertrag gibt?
 - ◆ Dies kann bereits zur Entwicklungszeit oder erst zur Laufzeit erkannt werden
 - ◆ Was ist, wenn der Vertrag nicht eingehalten wird?
 - ◆ Fallunterscheidung: Kunde vs Anbieter
 - ◆ Was mache ich mit einem gebrochenen Vertrag?
 - ◆ Alternativen: Abbrechen oder Weiterlaufen

Beispiel eines Fehlers

- ❖ Wenn $N = 0$, gibt es einen Fehler "Division-durch-0".

Schlechter Entwurf:
Mittelwert() ist nicht vor
 $N=0$ geschützt.

```
public int mittelwert (int arr[], int N) {  
    int avg = 0;  
    for (int k = 0; k < N; k++)  
        avg += arr[k];  
    avg = avg / N;  
    return avg;  
} // mittelwert( )
```

Was machen wir mit Fehlern?

Explizite Fehlerbehandlung im Programm

- ❖ **1. Möglichkeit:** Wir bauen die Fehlerbehandlung (hier: Fehlermeldung und Abbruch der Programmausführung) explizit in das Programm ein.
- ❖ 95% aller existierenden Systeme arbeiten so:-(

```
public class Statistik {
    public int mittelwert( int arr[], int N ) {
        int avg = 0;
        (
            if (N <= 0) {
                System.out.println("FEHLER: Division durch 0");
                System.exit(0);
            }
        )
        for (int k = 0; k < N; k++)
            avg += arr[k];
        avg = avg / N;
        return avg;
    } // mittelwert()
} // Statistik
```

Was machen wir mit Fehlern?

Spezifikation von Fehlersituationen (mit Verträgen)

- ❖ **2. Möglichkeit:** Wir schützen die Methode mit einem Vertrag

```
public class Statistik {  
    /**  
     * @Pre:  n > 0  
     */  
    public int mittelwert(int arr[], int n) {  
        int avg = 0;  
        for (int k = 0; k < n; k++)  
            avg += arr[k];  
        avg = avg / n;  
        return avg;  
    } // mittelwert()  
} // Statistik
```

- ❖ Ein Vertrag löst noch nicht das Problem der Fehlerbehandlung.
Was machen wir bei Vertragsbruch?

Was machen wir bei Vertragsbrüchen?

❖ **Wir verhindern sie (während der Entwicklung)**

- ◆ Wir beweisen, dass es keinen Vertragsbruch geben kann (*Zusicherungen*, Programmverifikation \Rightarrow Info II (später))
- ◆ Wir verbessern den Fehler im Modell (Revision des Entwurfs) oder im Quellprogramm (Testen & Debugging)

❖ **Wir leben mit ihnen (während der Laufzeit)**

- ◆ Wir melden den Fehler und stoppen das System
 - ◆ Mehr als 90% aller existierenden Programme machen das. Nicht zu empfehlen!
- ◆ Wir geben die Behandlung des Vertragsbruches an eine kompetente Stelle weiter (*Ausnahmen*)
 - \Rightarrow Info II (Java unterstützt Ausnahmen).
- ◆ Das System wird von einem fehlerhaften Zustand auf einen korrekten Zustand (zurück-)gesetzt (auch *defensive Programmierung* genannt)
 - ◆ Transaktionssysteme in Datenbanken (\Rightarrow Hauptstudium).

Ursachen für Vertragsbrüche

- ❖ Wenn ein Vertrag gebrochen wird, kann es dafür viele Ursachen geben:
- ❖ **Entwurfsfehler:**
 - ◆ Eine Invariante, Vor- oder Nachbedingung ist falsch formuliert
⇒ Vertrag revidieren
- ❖ **Benutzungsfehler:**
 - ◆ Der Vertrag wird durch den Klassen-*Benutzer* verletzt: Die Evaluierung der Vorbedingung ergibt **false**.
- ❖ **Implementierungsfehler:**
 - ◆ Der Methoden-*Implementierer* implementiert eine Methode, die die Nachbedingung verletzt:
Die Evaluierung der *Nachbedingung* ergibt **false**.
⇒ Implementierung verbessern
- ❖ Wir konzentrieren uns auf Benutzungsfehler (Verletzung der Vorbedingung) und ihre Behandlung durch Ausnahmen.

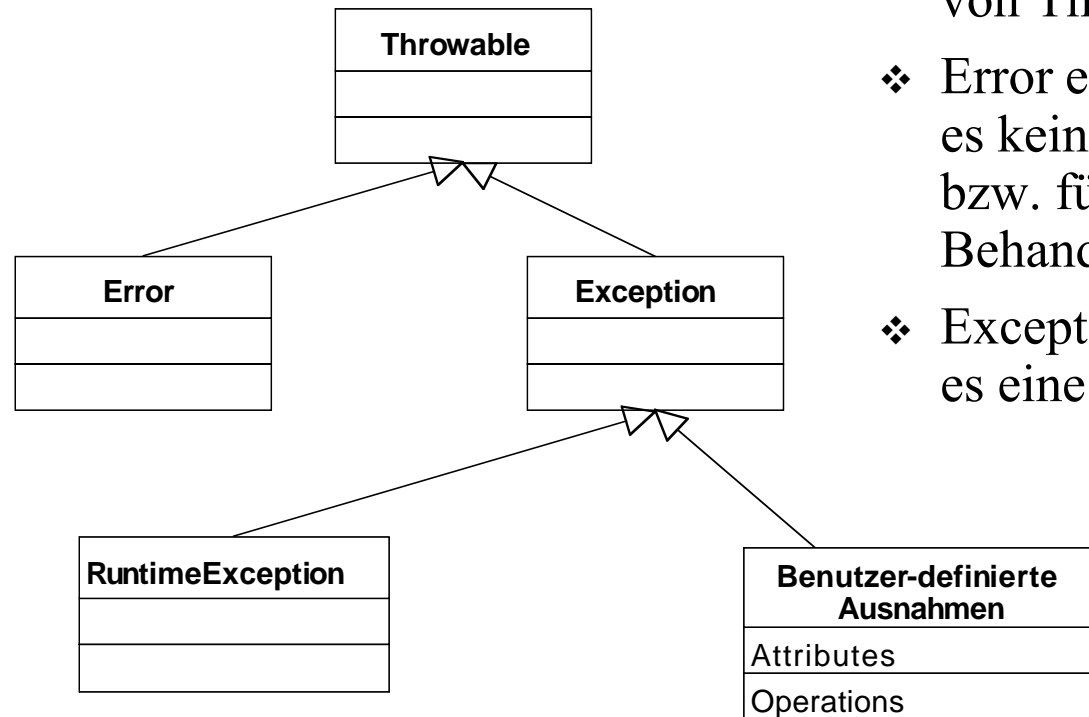
Ausnahmen

- ❖ **Definition Ausnahme (exception):** Ein Ereignis, das die normale Ausführungsreihenfolge in einem System unterbricht, da ein Fehler aufgetreten ist.
- ❖ **Definition Ausnahmebehandlung (exception handling):** Zur Behandlung von Ausnahmen wird zusätzlicher Programmcode bereitgestellt. Wenn eine Ausnahme auftritt, wird die Programmausführung unterbrochen, um die Ausnahmebehandlung durchzuführen.
- ❖ In Objekt-orientierten Sprachen (z.B. Java):
 - ◆ Jede Ausnahme ist Instanz einer Klasse, d.h. sie hat Methoden und Attribute, die die Unterscheidung verschiedener Fehlerursachen erlauben. Wenn diese Unterscheidbarkeit wichtig ist, bezeichnen wir die Klasse der Ausnahme auch als **Ausnahmetyp**.
 - ◆ Ein Ausnahme-Objekt enthält Einzelheiten über die Ursache seines Entstehens, die in der Ausnahmebehandlung genutzt werden können.
- ❖ Bei der Modellierung sind einzelne Ereignisse meist nicht von Interesse. Hier steht der Begriff der Ausnahme daher eher für den Ausnahmetyp.

Taxonomie von Ausnahmen

- ❖ Ein wesentlicher Teil des Entwurfs, besonders des detaillierten Entwurfs ist die Modellierung von Ausnahmen.
- ❖ Ziel der Modellierung ist die Erstellung einer Taxonomie, die es ermöglicht, Ausnahmen aus der Anwendungsdomäne und der Lösungsdomäne zu behandeln.
 - ◆ Ausnahmen sind gut für die Modellierung von potentiellen Problemen, z.B. bei der Eingabe von Werten in interaktiven Systemen.
- ❖ Die Programmiersprache Java stellt eine (erweiterbare) Taxonomie für Ausnahmen in Form einer Klassenhierarchie bereit.
 - ◆ Mithilfe von Spezialisierung können wir die modellierten Ausnahmen aus der Anwendungsdomäne und Lösungsdomäne als Unterklassen von dieser Ausnahmehierarchie definieren.


Java's Ausnahmhierarchie



- ❖ Jeder Ausnahmetyp ist Unterklasse von Throwable.
- ❖ Error enthält alle Ausnahmen, für die es keine Ausnahmebehandlung gibt bzw. für die keine sinnvolle Behandlung mehr möglich ist.
- ❖ Exceptions sind Ausnahmen, für die es eine Ausnahmebehandlung gibt.

*Spezifikation der Klasse **Exception***

```
public class Exception extends Throwable {  
    // Beschreibung der Ausnahme.  
    // Voreinstellung abhängig vom Ausnahmentyp  
    private String description;  
    // Konstruktoren  
    public Exception();  
    public Exception(String s); // s überschreibt  
                                // description  
    // Methoden  
    public String getMessage();  
}
```



Das Resultat von `getMessage()` ist die Beschreibung (**description**) der Ausnahme

Spezialisierung bei Ausnahmen

- ❖ Ausnahmen sind Klassen, wir können also Unterklassen definieren.
- ❖ **Beispiel:** Beim Prüfen von Eingaben wollen wir oft sicherstellen, dass der eingegebene Wert kleiner als ein maximaler Wert ist.

```
/**
 * Eine IntOutOfRange-Ausnahme tritt ein, wenn die eingegebene
 * ganze Zahl einen bestimmten Wert bound überschreitet.
 */
public class IntOutOfRange extends Exception {
    public IntOutOfRange () {}
    public IntOutOfRange (int bound) {
        super("Der Eingabewert ist größer als " + Bound);
    }
}
```

Im Konstruktor von **IntOutOfRange** wird mit **super()** der Konstruktor der Superklasse, also von **Exception**, aufgerufen.

Erzeugung von Ausnahmen in Java

- ❖ Wenn eine Ausnahme eintritt, wird ein entsprechendes Ausnahme-Objekt **"geworfen"** (**throw**).

```
/**
 * @pre n > 0
 */
public int mittelwert (int[] arr, int n) {
    int avg = 0;
    if (n <= 0)
        throw new Exception("Fehler: Division durch 0");
    for (int k = 0; k < N; k++)
        avg += arr[ k] ;
    avg = avg / N;
    return avg;
} // mittelWert()
```

- ❖ Diese Methoden-Implementierung ist noch nicht vollständig, denn
 - ◆ die Ausnahme muss innerhalb der Methode behandelt werden, oder
 - ◆ die Ausnahme muss vom Methodenaufrufer behandelt werden.

Ausnahmebehandlung in Java: try-catch-finally

- ❖ Ein "Versuchsblock" (**try**{...}) enthält Anweisungen, deren Ausführung *möglicherweise* eine *Ausnahme verursachen* kann. Mit einem Versuchsblock signalisieren wir unsere Bereitschaft, eventuell auftretende Ausnahmen zu behandeln.
- ❖ Die eigentliche Ausnahmebehandlung *fängt* die von einer Anweisung im Versuchsblock *geworfene Ausnahme* und besteht aus einem oder mehreren "Fang-Blöcken" (**catch**{...}):
 - ◆ Jeder Fang-Block ist für einen bestimmten Ausnahmetyp zuständig.
 - ◆ Die Definition von Fang-Blöcken ähnelt der Definition von Methoden, und enthält einen formalen Parameter (oft **e** genannt).
Der Typ des Parameters ist der Ausnahmetyp.
- ❖ Der "Final-Block" (**finally**{...}) macht die *Aufräumarbeiten*.
 - ◆ Die Angabe eines Final-Blocks ist nicht zwingend vorgeschrieben.
 - ◆ Ist ein Final-Block angegeben, so wird er nach dem Versuchsblock und der Ausnahmebehandlung ausgeführt, unabhängig davon, ob im Versuchs-Block eine Ausnahme erzeugt wurde oder nicht.

Benutzung des *try-catch-finally*-Rahmens in Java

Wir wissen, dass bei der Mittelwert-Berechnung eine Ausnahme erzeugt werden kann (falls $n \leq 0$), die wir behandeln wollen (*try*)

```
public int mittelwert( int arr[], int n ) {  
    int avg = 0;  
    try {  
        if ( n <= 0 )  
            throw new Exception("Fehler: Division durch 0");  
        for (int k = 0; k < N; k++)  
            avg += arr[k];  
        avg = avg / N;  
    }  
    catch (Exception e) {System.out.println(e.getMessage());}  
    finally {}  
    return avg;  
} // mittelWert()
```

Wenn $n \leq 0$ wahr ist, wird die Ausnahme instanziiert (*new Exception*) und geworfen (*throw*).

Hier wird das Ausnahme-Objekt *e* an die Ausnahmebehandlung übergeben (*catch*).

Der Final-Block (*finally*) könnte hier auch entfallen, da keine "Aufräumarbeiten" durchzuführen sind.

Behandlung von mehr als einem Ausnahmetyp

```
try {
    if ( /* Überprüfung Invariante, Vor- oder Nachbedingung */ )
        throw new ExceptionName1();
    if ( /* Überprüfung Invariante, Vor- oder Nachbedingung */ )
        throw new ExceptionName2();
    // ... weitere Überprüfungen
}
catch (ExceptionName1 ParameterName) {
    // Ausnahmebehandlung für ExceptionName1
}
catch (ExceptionName2 ParameterName) {
    // Ausnahmebehandlung für ExceptionName2
}
// ... weitere Ausnahmebehandlungen

finally {
    // Aufräumarbeiten
}
```

Deklaration von Ausnahmen

- ❖ Methoden können die Ausnahmebehandlung an den Aufrufer der Methode delegieren.
- ❖ Alle von einer Methode erzeugbaren Ausnahmen müssen in der Signatur mit dem Schlüsselwort **throws** deklariert sein.

- ❖ Allgemeine Form der Methodendeklaration:

```
ErgebnisTyp f ( Typ_1 param_1, ..., Typ_m param_m)  
    throws AusnahmeTyp_1, ... , AusnahmeTyp_n
```

- ❖ Die Ausnahmetypen **AusnahmeTyp_1, ..., AusnahmeTyp_n** sind logisch gesehen zusätzliche Ergebnisparametertypen (zu ErgebnisTyp)
- ❖ Ausnahmen gehören zur Schnittstelle einer Klasse und sind deshalb auch Teil des Vertrages:
 - ◆ Der Kunde darf nicht mit Ausnahmen überrascht werden, ohne zu wissen, wieso und woher sie kommen.
 - ◆ Der Anbieter des Vertrages muss also in der Schnittstelle spezifizieren, welche Ausnahmen die Methoden seiner Klassen erzeugen können.

Deklaration von Ausnahmen: Beispiel

```
public class Complex {  
    private double Re; // Realteil  
    private double Im; // Imaginärteil  
    ...  
    public Complex divide (Complex d)  
        throws DivisionbyZeroException {  
        ...  
        if ((d.Re == 0.0) && (d.Im ==0.0))  
            throw new DivisionbyZeroException();  
        ...  
    }  
}
```

throws *deklariert* die Ausnahme
DivisionbyZeroException

throw *erzeugt* die Ausnahme
DivisionbyZeroException

Deklaration von Ausnahmen: 2. Beispiel

- ❖ Wenn wir in einer Methode andere Methoden aufrufen, müssen wir wissen, welche Ausnahmen diese werfen können.
 - ◆ Diese Ausnahmen müssen wir entweder selbst behandeln oder an den Aufrufer unserer Methode weiterreichen (durch Deklaration).
- ❖ **Beispiel:** Die Methode **readLine()** der Klasse **BufferedReader** kann die Ausnahme **IOException** werfen.

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

```
throws IOException {
```

IOException muss deklariert werden...

```
    BufferedReader input = new BufferedReader  
        (new InputStreamReader(System.in));
```

```
    String inputString = input.readLine();
```

```
}
```

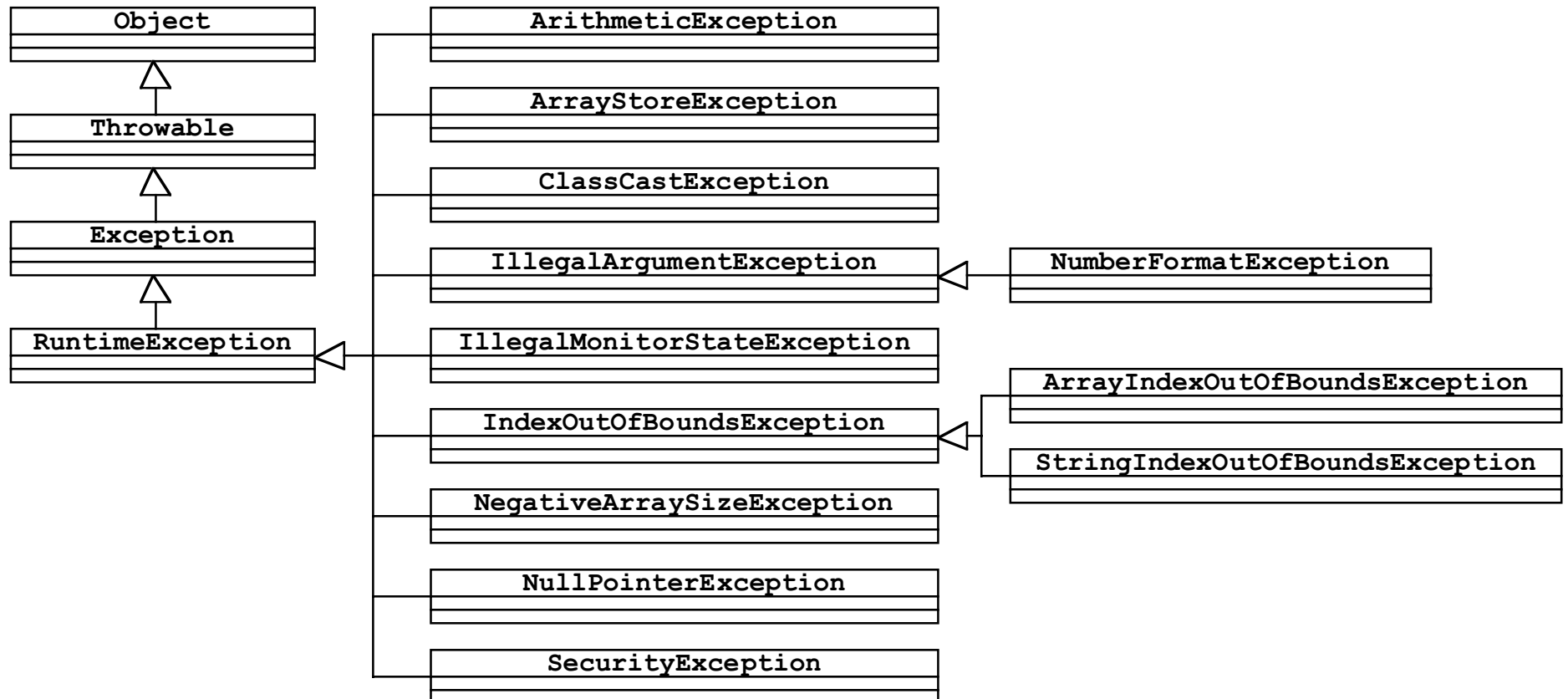
...denn **readLine()** kann sie werfen.

Geprüfte und Ungeprüfte Ausnahmen in Java

- ❖ Java unterscheidet zwischen ungeprüften und geprüften Ausnahmen.
- ❖ **Ungeprüfte Ausnahme** (unchecked exception): Eine Ausnahme, von der sich ein Programm normalerweise nicht erholen kann, und die man deshalb auch nicht prüfen muss.
 - ◆ Alle Ausnahmen vom Typ **Error**
- ❖ **Geprüfte Ausnahmen** (checked exception): Eine Ausnahme, von der sich ein Programm erholen kann, und die man deshalb behandeln muss:
 - ◆ Alle Klassen vom Typ **Exception**.
 - ◆ *"Jede Regel hat eine Ausnahme"*: Obwohl **RuntimeException** eine Unterklasse von **Exception** ist, können alle Ausnahmen vom Typ **RuntimeException** als ungeprüfte Ausnahmen betrachtet werden. **RuntimeException** wird implizit bei jeder Methodendefinition deklariert.

Ungeprüfte Ausnahmen in Java

- ❖ Ungeprüfte Ausnahmen heißen so, weil vom Java-Compiler nicht überprüft wird, ob sie deklariert sind.
 - ◆ Die Klasse **RuntimeException** und alle ihre Unterklassen sind ungeprüfte Ausnahmen.
 - ◆ Klassenhierarchie für **RuntimeException**:



Ungeprüfte Ausnahmen in Java: Laufzeit-Fehler

ArithmeticException

Division durch Null oder ein anderes arithmetisches Problem

ClassCastException

Ungültige Typ-Konvertierung eines Objektes in eine Klasse, von der es keine Instanz ist.

IllegalArgumentException

Methodenaufruf mit falschen Argumenten

NumberFormatException

Illegales Zahlenformat (z.B. beim Methodenaufruf)

IndexOutOfBoundsException

Ein Reihungs- oder Zeichenkettenindex ist außerhalb der Grenzen

ArrayIndexOutOfBoundsException

Ein Index in einer Reihung ist kleiner als 0 oder \geq als die Länge der Reihung

StringIndexOutOfBoundsException

Ein Zeichenkettenindex ist außerhalb der Grenzen

NullPointerException

Objektzugriff über **null**-Referenz

Wann kommen ungeprüfte Ausnahmen vor?

Beispiele (Java-Standardklassen)

<u>Klasse</u>	<u>Aufruf</u>	<u>Ausnahme</u>	<u>Beschreibung</u>
Double	valueOf(String)	NumberFormatException	String ist kein double
Integer	parseInt(String)	NumberFormatException	String ist kein int
String	String(String)	NullPointerException	String ist null
	indexOf(String)	NullPointerException	String ist null
	lastIndexOf(String)	NullPointerException	String ist null
	charAt(int)	StringIndexOutOfBoundsException	int ist ungültiger Index
	substring(int)	StringIndexOutOfBoundsException	int ist ungültiger Index
	substring(int,int)	StringIndexOutOfBoundsException	int ist ungültiger Index

Beispiel: Ausnahme-Erzeugung und Behandlung in Java

```
Public class Statistik {
    public int mittelwert( int arr[], int n )
        throws ArithmeticException {
        int avg = 0;
        if (n <= 0)
            throw new ArithmeticException("Fehler: Division durch 0");
        for (int k = 0; k < N; k++)
            avg += arr[k];
        avg = avg / N;
        return avg;
    } // mittelWert()
    public static void main(String[] args) {
        int numbers[] = {10, 20, 30, 30, 20, 10};
        try {
            Statistik stat = new Statistik();
            System.out.println( "Mittel: " +stat.mittelwert(numbers, 0) );
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage() );
            System.exit(0);
        }
    } // main()
} // Statistik
```

Hier wird die Ausnahme instanziiert und geworfen...

... und durch diesen Fang-Block behandelt.

Der Aufruf von `e.getMessage()` ergibt als Resultat die Beschreibung des Ausnahme-Objekts `e`

Voreingestellte Behandlung von Ausnahmen in main ()

- ❖ Wenn es im Hauptprogramm (**main ()**-Methode) keinen Fang-Block für eine Ausnahme gibt, dann behandelt das Laufzeitsystem von Java (die sog. *Java Virtual Machine (JVM)*) die Ausnahme.

```
public class Statistik {
    public int mittelWert(int arr[], int N) {
        int avg = 0;
        if (N <= 0)
            throw new ArithmeticException(" Fehler: Mitteln von 0 Elementen ");
        for (int k = 0; k < N; k++)
            avg += arr[k];
        avg = avg / N;
        return avg;
    } // mittelWert()
    public static void main(String args[]) {
        int numbers[] = {10, 20, 30, 30, 20, 10};
        Statistik stat = new Statistik();
        System.out.println("Mittel: " + stat.mittelWert(numbers, 0));
    } // main()
} // Statistik
```

Kein Fang-Block für **ArithmeticException**,
also übernimmt die JVM die Behandlung.

Ausgabe des Programms:

```
java.lang.ArithmeticException: Fehler: Mitteln von 0 Elementen
    at Statistik.mittelWert(Statistik.java:5)
    at Statistik.main(Statistik.java:14)
    at com.mw.Exec.run(JavaAppRunner.java:47)
```

Dokumentation von Ausnahmen in Javadoc

- ❖ Ausnahmen dokumentieren wir in Javadoc mit
@exception *class-name* *description*
 - ◆ Der **@exception**-Tag muss vor der Methode stehen, für die die Ausnahme deklariert wird.
 - ◆ ***class-name*** muss ein voll qualifizierter Name sein.
- ❖ Der Name der Ausnahme kommt in eine "*Throws*"-Sektion.
 - ◆ Eine "*Throws*"-Sektion wird nur für Methoden erzeugt, für die eine oder mehrere Ausnahmen deklariert sind.

Beispiel: Dokumentation von Ausnahmen

```
Public class Complex {
    private double Re; // Realteil
    private double Im; // Imaginärteil
    ...
    /**
     * Komplexe Division.
     * @param d Divisor: Für Division muss d != 0 sein
     * @exception DivisionbyZeroException: Wird erzeugt, wenn d == 0
     */
    public Complex divide (Complex d) throws DivisionbyZeroException {
        if (d.Re==0.0 && d.Im ==0.0)
            throw new DivisionbyZeroException();
        ...
    }
}
```

Deklaration der Ausnahme
DivisionbyZeroException

Implementation von Verträgen mit Ausnahmen

1. OCL-Spezifikations-Operationen implementieren wir in Java
2. Wir untersuchen alle Vor- und Nachbedingungen nach OCL-Prädikaten.
3. Für jedes neue OCL-Prädikat P , das wir in den Vor- und Nachbedingungen finden:
 - ◆ Wir deklarieren eine Java-Ausnahme \mathbf{J} .
 - ◆ Wir schreiben eine Anweisung, die die Gültigkeit des Prädikats überprüft:
 - ◆ Bei Vorbedingungen kommt die Überprüfung an den Anfang des Methoden-Rumpfes
 - ◆ Bei Nachbedingungen kommt die Überprüfung an das Ende des Methoden-Rumpfes
 - ◆ Für die Überprüfung wird das OCL-Prädikat in einen booleschen Ausdruck \mathbf{B} übersetzt.
 - ◆ Wenn ($\mathbf{B} == \mathbf{false}$) wahr ist, werfen wir die zugehörige Ausnahme \mathbf{J} .

Beispiel: Vertrag für AVL-Bäume

```
public class AVLTree {
    private AVLNode root;
    /**
     * @post result = root->isEmpty
     */
    public boolean isEmpty () {
        return (root == null);
    }
    /**
     * @pre  contains(key) = false
     * @post contains(key) = true
     * @post root.isAVL() = true
     */
    public void insert (int key) {
        ...
    }
}
```

```
public boolean contains (int key){
    ...
}
public AVLNode find (int key) {
    ...
}
/**
 * @pre contains(key) = true
 * @pre root.isAVL() = true
 * @post ...
 */
public void delete (int key) {
    ...
}
} // AVLTree
```

Beispiel: Implementation von isAVL ()

```
// Java-Implementierung der OCL-Spezifikationsoperation
private Boolean isAVL(AVLNode n) {
    if (n.leftChild.isEmpty() && n.rightChild.isEmpty())
        return true;
    else if (n.leftChild.isEmpty())
        return ((abs(n.hdiff) <= 1) && isAVL(n.rightChild));
    else if (n.rightChild.isEmpty())
        return ((abs(n.hdiff) <= 1) && isAVL(n.leftChild));
    else
        return ((abs(n.hdiff) <= 1) &&
                isAVL(n.leftChild) && isAVL(n.rightChild));
}
```

Bemerkung: Bei der Berechnung von **hdiff** wird die Höhe eines leeren Baum auf -1 festgelegt.

Beispiel: Implementation von delete ()

```
/**
 * @pre contains(key) = true
 * @pre root.isAVL() = true
 * @post ...
 * @exception NotContained Wird erzeugt, wenn key nicht im Baum ist.
 * @exception NotAVLTree Wird erzeugt, falls Baum kein AVL-Baum ist.
 */
public void delete (int key)
    throws NotContained, NotAVLTree {
    if (not contains(key))
        throw NotContained("Element " + key + " ist nicht im Baum");
    if (not isAVL(root))
        throw NotAVLTree("Der Baum ist gar kein AVL-Baum");
    ... // Der eigentlich Rumpf der delete() Operation...
} // delete()
```

Deklaration von 2 Ausnahmen:
NotContained und **NotAVLTree**

Beispiel: Ausnahmebehandlung für delete ()

- ❖ Mit den Ausnahmen wird die Behandlung des Vertragsbruches an den Benutzer weitergegeben.
- ❖ Im Falle verletzter Vorbedingungen ist der Benutzer in der Regel auch die kompetente Stelle, um adäquat zu reagieren.
- ❖ Im Beispiel wird der Aufruf von **delete ()** in der Methode **myDelete ()** in einen **try-catch**-Block eingebettet.

```
public void myDelete (AVLtree tree, int key) {  
    try { tree.delete(key); }  
    catch (NotContained e) {  
        // Dieser Vertragsbruch ist harmlos, wenn es nur darauf  
        // ankommt, dass key nicht mehr im Baum enthalten ist.  
        // Dann ist hier gar nichts zu tun.  
        // Er kann aber auf gefährliche Inkonsistenzen hinweisen  
        // (wenn key eigentlich im Baum enthalten sein müsste).  
        // Hier ist dann Ursachenforschung nötig.  
        ... // Code zur Behandlung von NotContained  
    } catch (NotAVLTree e) {  
        // Dieser Vertragsbruch deutet auf tiefe Inkonsistenzen hin.  
        // Eine Behandlung ist sicherlich schwierig ...  
        ... // Code zur Behandlung von NotAVLTree  
    }  
} // myDelete ()
```

Die Ausnahmen **NotContained** und **NotAVLTree** müssen nicht mehr deklariert werden.

Heuristiken für Entwurf von Ausnahmen

- ❖ **Defensiver Entwurf:** Versuchen Sie potentielle Probleme zu sehen, die die normale Folge von Ereignissen verändern.
 - ◆ Ein guter Ausgangspunkt sind Anwendungsfälle. Spezielle Anwendungsfälle sind gute Kandidaten für Ausnahmen.
 - ◆ Auch die inkorrekte Eingabe von Werten bei interaktiven Systemen lässt sich oft gut als Ausnahme modellieren.
- ❖ **Lokale Behandlung:** Versuchen Sie, Ausnahmen möglichst lokal, d.h. "in der Nähe" ihres Auftretens, zu behandeln.
- ❖ **Ausnahmen in Klassenbibliotheken:** Schreiben Sie auch in der `main ()`-Methode Fang-Blöcke für alle Ausnahmen, die Methoden aus Klassenbibliotheken werfen könnten, sonst überlassen Sie Ihr Schicksal den oft unverständlichen Fehler-Beschreibungen der JVM.
- ❖ **Ausnahmen beschreiben Ausnahme-Situationen:** Benutzen Sie Ausnahmen nicht für die Behandlung von "normalen" Situationen (⇒ Vorlesungsblock "*Ereignis-basierte Programmierung*")

Zusammenfassung

- ❖ Ausnahmen sind Objekte (Instanzen von Ausnahmetypen/-klassen)
- ❖ Behandlung von Ausnahmen: **try-catch-finally**-Rahmen
- ❖ Geprüfte und ungeprüfte Ausnahmen:
 - ◆ Alle geprüften Ausnahmen müssen deklariert oder behandelt werden.
- ❖ Benutzer-definierte Ausnahmen sind normalerweise Spezialisierungen der Klassen **Exception**
- ❖ Ausnahmen sind ein gutes Werkzeug, um die Verletzung von Verträgen während der Programmausführung zu melden bzw. zu behandeln.