TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Ausnahmebehandlung mit Exceptions

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Ausnahmebehandlung I

Wie im Abschnitt zur "Terminierung" bereits angesprochen, kann die Auswertung eines Ausdrucks bzw. die Ausführung einer Anweisung:

- normal terminieren
- in eine Ausnahmesituation kommen und abrupt terminieren
- nicht terminieren

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Ausnahmebehandlung II

Es gibt drei Arten von Ausnahmesituationen:

1. Vom Programmierer schwer zu kontrollierende und zu beseitigende Situtationen (z.B. Speichermangel)
2. Programmierfehler (z.B. Null-Dereferenzierung, Verletzung von Indexgrenzen)
3. Zeitweise nicht verfügbare Ressourcen, anwendungsspezifische Ausnahmen, die behebbar sind.

Ausnahmen werden in Programmiersprachen verschieden behandelt:

- Programmabbruch (engl. *abort*)
- Ausnahmebehandlung

Ausnahmebehandlung in Java

Java bietet Sprachmittel für die **Ausnahmebehandlung** (engl. **exception handling**).

Dabei spielen drei Aspekte eine Rolle:

1. Wann/wie werden Ausnahmen ausgelöst?
2. Wie kann man sie abfangen?
3. Welcher Ausnametyp ist passend bzw. wie kann man neue Ausnahmetypen deklarieren?

Auslösen von Ausnahmen

Das Auslösen einer Ausnahme kann

- sprachdefiniert (z.B. NullPointer, IndexOutOfBounds) oder
- durch eine Anweisung spezifiziert sein.

In Java gibt es zum Auslösen von Ausnahmen die `throw`-Anweisung.

throw-Anweisung

Syntax:
Die `throw`-Anweisung hat die Form:

```
throw <Ausdruck>;
```

wobei der Ausdruck ein Ausnahmeobjekt als Ergebnis liefern muss.

Semantik:
Werte den Ausdruck aus.

Löst die Auswertung eine Ausnahme aus, ist dies die Ausnahme, die von der Anweisung ausgelöst wird.

Andernfalls löse die Ausnahme aus, die das Ergebnis des Ausdrucks ist.

Abfangen von Ausnahmen I

Die try-catch-Anweisung dient dem Abfangen und Behandeln von Ausnahmen:

```
3  void myMethod (String[] sfeld) {
4     try {
5       System.out.println(sfeld[0]);
6       System.out.println(sfeld[1]);
7    } catch (NullPointerException e) {
8       System.out.println("sfeld is null");
9    } catch (IndexOutOfBoundsException e){
10      System.out.println("sfeld too small");
11   }
12 }
```

Abfangen von Ausnahmen II

Tritt eine Ausnahme vom Typ T im try-Block auf, wird ein T-Objekt `x` erzeugt. Ist der Typ T in der Liste der catch-Klauseln aufgeführt,

- wird die Ausnahme *gefangen*,
- x an den Bezeichner der entsprechenden catch-Klausel gebunden und
- diese `catch`-Klausel ausgeführt (Verfeinerung in späterem Abschnitt).

Benutzerdefinierte Ausnahmetypen

Die Deklaration von Exception-Klassen behandeln wir in einem späteren Abschnitt.

Bemerkung:
Java verlangt die Deklaration bestimmter Ausnahmetypen in der Signatur einer Methoden m, wenn sie nicht von m abgefangen werden.

Beispiel:

```
int m(int i) throws SomeException {
   if(i < 0) {
      throw new SomeException();
   }
   ...
}
```


Beispiel: Ausnahmebehandlung

```
3  public static void main(String[] args) {
4   try {
5    int ergebnis = 0;
6    int m = Integer.parseInt(args[0]);
7    int n = Integer.parseInt(args[1]);
8
9    long aux = (long) m + (long) n;
10   if (aux > Integer.MAX_VALUE) {
11       throw new UeberlaufException(); // selbstdefiniert
12   }
13   ergebnis = (int) aux;
14  } catch (IndexOutOfBoundsException e) {
15      System.out.println("Zuwenig Argumente");
16  } catch (NumberFormatException e) {
17      System.out.println("Parameter ist keine int-Konstante");
18  } catch (UeberlaufException e) {
19      System.out.println("Ueberlauf bei Addition");
20  }
21 }
```


Beispiel: Stacktraces

```
3  public static void main(String[] args) throws Exception {
4      double[] a = {};
5      System.out.println("avg = " + average(a));
6  }
7  static double average(double[] a) throws Exception {
8      checkPreconditions(a);
9      int sum = 0;
10     for (int i = 0; i < a.length; i++) {
11         sum += a[i];
12     }
13     return sum / a.length;
14 }
15 static void checkPreconditions(double[] a) throws Exception {
16     if (a == null) {
17         throw new Exception("Array darf nicht null sein");
18     }
19     if (a.length == 0) {
20         throw new Exception("Array darf nicht leer sein");
21     }
22 }
```

Ausgabe im Fehlerfall

Die Ausführung führt zu einem Programmabbruch, weil die Ausnahme nicht in einem `try`-Block behandelt wurde.

Als Fehlermeldung wird ein sogenannter *Stacktrace* angezeigt:

```
Exception in thread "main" java.lang.Exception: Array darf nicht leer sein
    at ExceptionTest.checkPreconditions(ExceptionTest.java:20)
    at ExceptionTest.average(ExceptionTest.java:8)
    at ExceptionTest.main(ExceptionTest.java:5)
```


Struktur eines Stacktraces

- Erste Zeile: Fehlermeldung mit Klassenname der Ausnahme (hier: `java.lang.Exception`)
- Folgende Zeilen: Liste der Methoden-Aufrufe, die beim Werfen der Ausnahme aktiv waren
- Jede weitere Zeile hat die Form:
  `at Klassenname.Methodenname(Dateiname:Zeilennummer)`
- Die Zeilennummern geben jeweils die aktuelle Position im Programm zum Zeitpunkt der Ausnahme an.

Kapselung und Strukturieren von Klassen

Kapselung und Strukturieren von Klassen

Zwei Aspekte zur weiteren Strukturierung objektorientierter Programme:

- Schnittstellenbildung und Kapselung
- Schachtelung von Klassen und Paketen

Schnittstellenbildung und Kapselung

Objekte stellen eine bestimmte Funktionalität/Dienste zur Verfügung:

- aus Anwendersicht: Nachrichten schicken, Ergebnisse empfangen
- aus Implementierungssicht: Realisierung der Zustände und Funktionalität durch
  - → objektlokale Attribute
  - → Referenzen auf andere Objekte
  - → Implementierung von Methoden

Ziel:

- Lege die Anwendungsschnittstelle genau fest.
- Verhindere Zugriff "von außen" auf Implementierungsteile, die nur für internen Gebrauch bestimmt sind.

Begriffsklärung: Anwendungsschnittstelle

Die **Anwendungsschnittstelle** eines Objekts bzw. eines Referenztyps besteht aus

- den Nachrichten, die es für Anwender zur Verfügung stellt;
- den Attributen, die für den direkten Zugriff von Anwendern bestimmt sind.

Bemerkung:

- Die Festlegung von Anwendungsschnittstellen ist eine Entwurfsentscheidung.
- Direkter Zugriff auf Attribute muss nicht gewährt werden, sondern kann mit Nachrichten/Methoden realisiert werden.

Beispiel

```
class MitDirektemAttributZugriff {
   public int attr;
}

class AttributZugriffUeberMethoden {
   private int attr;
   public int getAttr() {
    return attr;
   }
   public void setAttr(int a) {
    attr = a;
   }
}
```


Zugriffsfunktionen

- Der Zugriff auf Attribute wird häufig über spezifische *getter-* / *setter-*Methoden realisiert.
- Dies erlaubt es insbesondere Modifikationen zu kontrollieren, beispielsweise
  - das Überprüfen der Gültigkeit von neuen Attributwerten
  - das Benachrichtigen von anderen Objekten bei Änderungen (⇒ späteres Kapitel zu "Beobachtermuster")
- Man kann so ausserdem erzwingen, dass Attributwerte nicht direkt verändert werden können, in dem z.B. keine `set`-Methode zur Verfügung gestellt wird.

Aufgabe

1. Schreiben Sie die Klasse `Date` so um, dass die Felder nicht direkt zugreifbar sind.
2. Fügen Sie nun Getter-Methoden für die einzelnen Attribute hinzu.
3. Warum ist es sinnvoll keine beliebigen Veränderungen bei den Datumsobjekten zu erlauben?

```
class Date {
    int day;
    int month;
    int year;

    Date(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
    }
}
```


Begriffsklärung: Information Hiding

Das Prinzip des **Information Hiding** (deutsch meist **Geheimnisprinzip**) besagt, dass

- Anwendern nur die Informationen zur Verfügung stehen sollen, die zur Anwendungsschnittstelle gehören,
- alle anderen Informationen und Implementierungsdetails für ihn verborgen und möglichst nicht zugreifbar sind.

Gründe für Information Hiding:

- Vermeiden unsachgemäßer Anwendung
- Vereinfachen von Software durch Reduktion der Abhängigkeiten zwischen ihren Teilen
- Austausch von Implementierungsteilen

Beispiele

- **Postleitzahlen**: In (West-)Deutschland wurden bis 1962 zweistellige, danach vierstellige Postleitzahlen; seit 1993 schließlich fünfstellige. Es ist semantisch nicht sinnvoll mit Postleitzahlen zu rechnen. In vielen Ländern bestehen die Postleitzahlen aus bis zu 10 Zeichen und enthalten nicht nur Ziffern, sondern auch Buchstaben und Bindestriche.
- **IP-Adressen**: Das Internetprotokoll (IP) ist eines der wichtigsten Protokolle des Internets. Es legt fest, wie Geräte eindeutig addressiert werden können. Die Version IPv4 verwendet 32-Bit-Adressen und erlaubt es daher, Netze von bis zu 4.3 Milliarden direkt addressierbaren Geräten zu verwalten. Die Nachfolgeversion IPv6 nutzt 128-Bit-Adressen und erlaubt die direkte Adressierung von etwa $3,4 \cdot 10^{38}$ Geräten.

Information Hiding in Java

Java ermöglicht es für Programmelemente sogenannte *Zugriffsbereiche* zu deklarieren.

Vereinfachend gesagt kann ein Programmelement nur innerhalb seines Zugriffsbereichs verwendet werden.

Java unterscheidet vier Arten von Zugriffsbereichen:

- nur innerhalb der eigenen Klasse (Modifikator `private`)
- nur innerhalb des eigenen Pakets (ohne Modifikator)
- nur innerhalb des eigenen Pakets und in Subklassen (Modifikator `protected`)
- ohne Zugriffsbeschränkung (Modifikator `public`)

Generell können Klassen, Attribute, Methoden und Konstruktoren mit diesen Zugriffsmodifikatoren deklariert werden.

Beispiel: Vermeiden falscher Anwendung I

```
public class RunnersDiary {
    private String name;
    private Node first;
    public RunnersDiary(String name) {
        this.name = name;
    }
    // Liefert den Namen des Laeufers
    public String getName() {...}

    // Fuegt einen neuen Eintrag hinzu
    public void add(Entry e) {...}

    // Entfernt alle Eintraege zu einem Datum
    public void remove(Date d) {...}

    // Liefert das Element an Position index
    public Entry get(int index) {...}

    // Darstellung als String
    public String toString() {...}
}
```


Beispiel: Vermeiden falscher Anwendung II

Die Schnittstelle erlaubt keinen direkten Zugriff auf die Knoten der `Entry`-Liste.

- Es gibt keine Methode `getFirst` o.ä., und keine der Methoden liefert eine Referenz auf ein `Node`-Objekt.
- Außerhalb der Klasse `RunnersDiary` kann daher weder das Attribut `first` noch die Listenstruktur verändert werden.
- Damit wird vermieden, dass die Listen beispielsweise Zyklen enthält.
- Es erlaubt ausserdem die Implementierung der `Node`-Klasse beliebig abzuändern, ohne dass Code, der Objekte der Klasse `RunnersDiary` verwendet, abgeändert werden muss.

Beispiel: Austausch von Implementierungen I

```
1  public class RunnersDiary {
2      private String name;
3      private int size;
4      private Node first;
5
6      RunnersDiary(String name) {
7          this.name = name;
8          this.size = 0;
9          this.first = null;
10     }
11
12     // Liefert die Anzahl der Eintraege
13     String getSize() {
14       return this.size;
15     }
16
17     // Fuegt einen neuen Eintrag hinzu
18     void add(Entry e) {
19       ...
20       this.size++;
21     }
22     ...
23 }
```


Beispiel: Austausch von Implementierungen II

Die Schnittstelle erlaubt keinen direkten Zugriff auf die Länge der `Entry`-Liste.

- Außerhalb der Klasse `RunnersDiary` kann nur mittels `getSize()` die Anzahl der Listeneinträge abgefragt werden.
- Verwender der Klasse können das `size`-Attribut nicht beliebig verändern.
- Damit wird vermieden, dass es Inkonsistenzen zwischen der Listenstruktur und dem Attribut gibt.

Beispiel: Web-Seiten I

Objekte zur Repräsentierung trivialer Web-Seiten mit Titelzeile und Inhalt

```
1  public class W3Seite {
2     private String titel;
3     private String inhalt;
4
5     public W3Seite(String t, String i) {
6        titel  = t;
7        inhalt = i;
8     }
9     public String getTitel() {
10     return titel;
11    }
12    public String getInhalt(){
13     return inhalt;
14    }
15 }
```


Beispiel: Web-Seiten II

Die obige Klasse kann ersetzt werden durch die folgende, ohne dass Anwender der Klasse davon betroffen werden:

```
1  public class W3Seite {
2    private String seite;
3    public W3Seite( String t, String i ) {
4      seite =  "<TITLE>" + t + "</TITLE>" + i ;
5    }
6    public String getTitel() {
7      int ix = seite.indexOf("</TITLE>") - 7;
8      return new String(seite.toCharArray(), 7, ix);
9    }
10   public String getInhalt() {
11     int ix = seite.indexOf("</TITLE>") + 8;
12     return new String(seite.toCharArray(), ix,
13                       seite.length() - ix );
14   }
15 }
```

Bemerkung

- Information Hiding erlaubt insbesondere:
  - konsistente Namensänderungen in versteckten Implementierungsteilen
  - Verändern versteckter Implementierungsteile, soweit sie keine Auswirkungen auf die öffentliche Funktionalität haben (kritisch)

    Beispiel:
    Die zweite Implementierung von W3Seite kann nur dann anstelle der ersten benutzt werden, wenn Titel den Substring </TITLE> nicht enthalten.
- Attribute sollten privat sein und nur in Ausnahmefällen öffentlich.

Begriffsklärung: Kapselung

Wir verstehen unter **Kapselung** eine verschärfte Form des Information Hiding, die es dem Anwender unmöglich macht, auf interne Eigenschaften gekapselter Objekte zuzugreifen.

Bemerkung:

- Die Verwendung von `private` führt nicht automatisch zur Kapselung.
- Um Kapselung zu erreichen, bedarf es eines Zusammenwirkens unterschiedlicher Sprachmittel und Programmiertechniken, die in Java sehr komplex sein kann.

Beispiel: Kapselung in Java

Der private Zugriffsbereich bezieht sich auf die Klasse und im Allg. nicht auf einzelne Objekte:

```
public class FamilienMitglied {
   private int a;
   private FamilienMitglied geschwister;

   public void incrA(){
      a++;
      geschwister.a++;
   }
}
```

Die Methode `incrA` modifiziert nicht nur das Objekt, auf dem sie aufgerufen wurde, sondern auch das private Attribut `a` des Geschwisterobjekts.

Character

Datentyp `char`

- Der Basisdatentyp `char` repräsentiert einzelne Zeichen aus dem Unicode-Zeichensatz, z.B. Ziffern, Groß- und Kleinbuchstaben, Satzzeichen, White space (Leerzeichen, Tabularzeichen, Zeilenumbruchzeichen), etc.

Typbezeichner	`char`
Werte	Zeichen
Typische Literale	`'a', '7', 'B'`

- Methoden aus der `Character`-Bibliothek

`boolean isLetter(char ch)`	Testet, ob das Zeichen ein Buchstabe ist
`boolean isDigit(char ch)`	Testet, ob das Zeichen eine Ziffer ist
`boolean isUpperCase(char ch)`	Testet, ob das Zeichen ein Großbuchstabe ist
`String toString(char ch)`	Liefert ein String-Objekt

Darstellung von `char`-Werten

- `char`-Werte repräsentieren die Unicode-Zeichen eigentlich Zahlen zwischen 0 und 65535 (16-bit).
  Zum Beispiel: `'a' == 97`, `'b' == 98`, `'A' == 65`, `'B' == 66`, `'0' == 48` und `'1' == 49`.
- Man kann sie daher vergleichen und auch mit ihnen rechnen:

```
boolean istBuchstabe = (c >= 'a' && c <= 'z')
                     || (c >= 'A' && c <= 'Z');

// Alle Zeichen von '0' bis 'z' ausgeben:
for (char c = '0'; c <= 'z'; c++) {
  System.out.println((int) c + ": " + c);
}

// Umwandlung von Zeichen '3' zu Zahl 3:
char c = '3';
int i = c - '0';
```


Spezielle Zeichen

`'\"'`	doppeltes Anführungszeichen
`'\''`	einfaches Anführungszeichen
`'\n'`	Zeilenumbruch
`'\t'`	Tabulatorzeichen
`'\b'`	Backspace
`'\\'`	Backslash-Zeichen

Diese Zeichen können auch innerhalb von Strings verwendet werden.

Umwandlung zwischen Strings und Charactern

- Der Konstruktor `String(char[] value)` liefert ein String-Objekt mit den Zeichen des Arrays `value`.
- Der Konstruktor `String(char[] value, int offset, int count)` liefert ein String-Objekt der Länge `count` mit den Zeichen aus dem Array `value`, ab der Position `offset`.
- Die Methode `toCharArray()` liefert für ein String-Objekte ein `char`-Array, dessen Einträge den Zeichen des Strings entsprechen.
- Die Methode `charAt(int index)` liefert den Character an der Position `index` in einem String.
- Die Methode `indexOf(int ch)` liefert die Position des ersten Vorkommens des Characters `ch` in einem String.

Iteratoren

Iteratoren

Iteratoren erlauben es, schrittweise über Behälterdatenstrukturen wie Listen zu laufen, so dass alle Elemente der Reihe nach besucht werden.

Im Zusammenhang mit Kapselung sind sie unverzichtbar.

```
public class RunnersDiaryIterator {
    //prueft, ob es noch weitere Eintraege gibt
    public boolean hasNext(){ ... }

    //liefert den naechsten Eintrag
    public Entry next() {...}
}
```


Beispiel: Iterator für Lauftagebuch I

Wir reichern die Klasse `RunnersDiary` mit Iteratoren an:

```
public class RunnersDiary {
  // Erweiterung um Iteratoren
  private Node first;
  ...
  public RunnersDiaryIterator iterator() {
     return new RunnersDiaryIterator(first);
  }
}
```


Beispiel: Iterator für Lauftagebuch II

```
1  import java.util.NoSuchElementException;
2
3  public class RunnersDiaryIterator {
4    private Node current;
5    public RunnersDiaryIterator(Node e) {
6      this.current = e;
7    }
8    //prueft, ob es noch weitere Eintraege gibt
9    public boolean hasNext(){
10     return current != null;
11   }
12   //liefert den naechsten Eintrag
13   public Entry next() {
14     if (current == null) {
15        throw new NoSuchElementException();
16     }
17     int res = current.getEntry;
18     current = current.getNext();
19     return res;
20   }
21 }
```

Beispiel: Iterator für Lauftagebuch III

```
1  public class RunnersDiaryTest {
2    public static void main(String[] args) {
3
4      RunnersDiary diary = new RunnersDiary("Hugo");
5
6      diary.add(new Entry(new Date(2,3,2015),5,28,"frisch"));
7      diary.add(new Entry(new Date(5,7,2015),8.2,40,"k.o."));
8      diary.add(new Entry(new Date(8,9,2015),10.4,54,"muede"));
9
10     RunnersDiaryIterator iter = diary.iterator();
11     while(iter.hasNext()) {
12       StdOut.println(iter.next().getDate());
13     }
14   }
15 }
```

Bemerkung:

Der Iterator muss Zugriff auf die interne Repräsentation der Datenstruktur haben, über die er iteriert.