Java I – Vorlesung 8
Exceptions

14.6.2004

Exceptions (Ausnahmen)
Klassen für Ein- und Ausgabe

Ausnahmen

- Ausnahmen (Exceptions) sind ein Mechanismus zum kontrollierten Melden und Reagieren auf Fehler.
- Man kann Ausnahmen werfen und muss sie irgendwo fangen.
- Wir mussten bis jetzt um Ausnahmen herumarbeiten -- jetzt lernen wir, wie sie richtig gehen!

Ausnahmen: Motivation

- Man kann Fehler in Methoden durch definierte Rückgabewerte anzeigen (z.B. null).
- Manchmal ist es aber furchtbar umständlich, auf mögliche Fehlerwerte zu testen.
- Manchmal gibt es keine sinnvollen Rückgabewerte.
- Compiler kann uns zwingen, uns beim Programmieren mit möglichen Fehlern auseinanderzusetzen.

Ausnahmen: Ein Beispiel

```
// IntList, IntNil aus Übung 4

IntList list = new IntNil();

int meineZahl = list.first(); // FEHLER!
```

Ausnahmen: Ein Beispiel

```
class IntNil {
  public int first() {
    // nehmen wir mal an, -1 wäre kein
    // gültiger Wert.
    return -1;
  }
}

IntList list = new IntNil();
int meineZahl;

meineZahl = list.first();
if( meineZahl != -1 ) {
  ...
}
```


Ausnahmen: Ein Beispiel

```
import java.util.*;

class IntNil {
  public int first()
    throws NoSuchElementException {
    throw new NoSuchElementException();
  }
}

IntList list = new IntNil();

try {
  int meineZahl = list.first();
} catch(NoSuchElementException e) {
  System.err.println("no such element!");
}
```

Ausnahmen: Ein zweites Beispiel

```
liesDatei {
  öffne die Datei;
  berechne ihre Größe;
  reserviere Speicher zum Einlesen;
  lies Datei ein;
  schließe Datei;
}
```


Ausnahmen: Ein zweites Beispiel

```
fehlerCodeType liesDatei {
  öffne die Datei;
  if( Datei wurde erfolgreich geöffnet ) {
    berechne ihre Größe;
    if( Größe wurde erfolgreich berechnet ) {
      reserviere Speicher zum Einlesen;
      if( Speicher wurde reserviert ) {
        lies Datei ein;
        if( Fehler beim Einlesen )
          fehlerCode = -4;
        else
          fehlerCode = 0;
      } else
        fehlerCode = -3; // Speicher reservieren
    } else
      fehlerCode = -2;  // Größe berechnen
    schließe Datei;
  } else
    fehlerCode = -1; // Datei öffnen
}
```

Ausnahmen: Ein zweites Beispiel

```
liesDatei {
  try {
    öffne die Datei;
    berechne ihre Größe;
    reserviere Speicher zum Einlesen;
    lies Datei ein;
    schließe Datei;
  } catch( Fehler beim Öffnen der Datei ) {
    ...
  } catch( Fehler beim Berechnen der Größe ) {
    ...
  } catch( Fehler beim Reservieren des Speichers ) {
    ...
  } catch( Fehler beim Einlesen ) {
    ...
  }
}
```


Ausnahmen: Syntax

- Ausnahmen werden mit einer `throw`-Anweisung geworfen.
- Ausnahmen, die in einem `try`-Block auftreten, können mit einem `catch`-Block gefangen werden.
- Im Kopf des `catch`-Blocks steht ein Exception-Parameter, der im Körper verwendet werden kann.
- Ein `try`-Block kann mehrere `catch`-Blöcke haben.

Wer fängt Ausnahmen?

- Wenn Ausnahme geworfen wird, wird die normale Verarbeitung sofort abgebrochen.
- Die VM sucht dann nach einem `catch`-Block, der die geworfene Ausnahme fängt, und setzt die Verarbeitung dort fort.
- Die Ausnahme wird vom tiefsten passenden aktiven `try-catch`-Block gefangen.
- Die `catch`-Blöcke des gleichen try müssen gegenseitig inkompatibel sein (d.h. höchstens ein Block passt).

Wer fängt Ausnahmen?

- Wenn Ausnahme innerhalb von einem `try-catch`-Block geworfen wird, versucht dieser Block sie zu fangen.
- Wenn das nicht klappt, gehen wir aktive Methodenaufrufe entlang, bis wir einen passenden `try-catch`-Block finden.
- Wenn niemand die Ausnahme fängt, wird das Programm abgebrochen.

Wer fängt Ausnahmen: Ein Beispiel

```
class IntNil {
  public int first()
    throws NoSuchElementException {
    throw new NoSuchElementException();
  }
  public void iterate()
     throws NoSuchElementException {
   // ...
    ... first() ...
  }
  public static void main(String[] args) {
    try {
       ... iterate() ...
    } catch(NoSuchElementException e) {
    // Exception aus first() wird hier gefangen
    }
}
```


Ausnahmen als Objekte

- Technisch ist jede Ausnahme ein Objekt, das Instanz einer von `Throwable` abgeleiteten Klasse ist.
- Es gibt drei Arten von Throwables:
  - checked exceptions: der Normalfall.
  - Laufzeit-Exceptions: müssen nicht gefangen werden
  - Fehler: tiefe böse Fehler, sollen nicht gefangen werden.

Checked Exceptions

- Checked Exceptions sind Objekte einer Klasse, die von `Exception` abgeleitet ist.
- Wenn eine Methode (durch `throw` oder Aufruf einer anderen Methode) eine checked exception erzeugen kann, die sie nicht direkt fängt, muss sie sie deklarieren:

```
ReturnType meth(...) throws A, B
```

- `NoSuchElementException` (s.o.) ist eine Checked Exception.

Checked Exceptions: Ein Beispiel

```
class IntNil {
  public int first() {
    // illegal: muss Exception deklarieren!
    throw new NoSuchElementException();
  }
}
```

Checked Exceptions: Ein Beispiel

```
class IntNil {
  public int first()
    throws NoSuchElementException {
    throw new NoSuchElementException();
  }
  public void iterate() {
    // illegal: first könnte Exception werfen,
    // die hier nicht gefangen wird.
    // ...
    ... first() ...
  }
}
```


Checked Exceptions: Ein Beispiel

```
class IntNil {
  public int first()
    throws NoSuchElementException {
    throw new NoSuchElementException();
  }
  public void iterate()
     throws NoSuchElementException {
   // ...
    ... first() ...
  }
}
```

Fangen mit Subtyping

- Ein Exception-Objekt o lässt sich von einem catch-Block für Klasse E fangen, wenn o eine Instanz von E ist (wie mit instanceof).
- Das bedeutet: Ein catch-Block fängt auch alle spezielleren Exceptions mit.
- Man kann damit gleichartige Exceptions mit dem gleichen Code verarbeiten.

Fangen mit Subtyping: Ein Beispiel

```
class IntNil {
  public int first()
    throws NoSuchElementException {
    throw new NoSuchElementException();
  }
}

try {
  IntNil liste = new IntNil();
  int x = liste.first();
} catch(Exception e) {
  // Dieser Block fängt die
  // NoSuchElementException ab
}
```

Eigene Exception-Klassen

- Man kann eigene Klassen für Ausnahmen deklarieren, indem man sie von Exception ableitet.
- Das ist sinnvoll, wenn man
  - feinere Unterscheidungen bzgl. Exception-Typ treffen will
  - sich von vordefinierten Exception-Typen unabhängig machen will.

Eigene Exception-Typen: Ein Beispiel

```
class EmptyListException extends Exception {
  EmptyListException() {
    super();
  }
  EmptyListException(String s) {
    super(s);
  }
}

class IntNil {
  public int first()
    throws ÉmptyListException {
    throw new EmptyListException();
  }
}
```

Laufzeit-Ausnahmen

- Manche Ausnahmen können so häufig auftreten, dass es lästig wäre, sie alle abfangen zu müssen, z.B.
  - `ArrayOutOfBoundsException`
  - Teilen durch Null
  - Feld- oder Methodenselektion auf null
- Solche Ausnahmen sind als Laufzeit-Ausnahmen realisiert und müssen nicht gefangen oder deklariert werden (können aber).
- Technisch: Abgeleitet von `RuntimeException`.

Laufzeit-Ausnahmen: Ein Beispiel

```
try {
  int[] array = {1,2,3};
  int x = array[5];
} catch(ArrayOutOfBoundsException e) {
  // Vorsicht: Das hier ist schlechter Stil!
  // ...
}
```

Fehler

- Die dritte Art von Throwables sind Fehler (Unterklassen von `Error`).
- Fehler zeigen tiefliegende Probleme in der VM an, z.B. dass Linker eine Klasse nicht gefunden hat.
- Typischerweise werden Errors nicht gefangen, sondern das Programm wird beendet.
- Errors werden nicht in `throws`-Klausel deklariert.

Ausnahmen und Vererbung

- Wenn man eine Methode überschreibt, muss die überschreibende Methode weniger und/oder speziellere Exceptions in throws-Klausel haben.
- Zweck: Compiler hat überprüft, dass try-catch-Block für überschriebene Implementierung vollständig sind. Vollständigkeit muss auch für überschreibende Methode gelten.

finally

- An einen `try-catch`-Block darf man auch einen `finally`-Block anhängen.
- Ein `finally`-Block wird auf jeden Fall nach Ausführung des `try`-Blocks ausgeführt -- sogar wenn `try`-Block durch Exception abgebrochen wurde.
- `finally`-Block wird direkt nach seinem eigenen, aber vor allen höheren `catch`-Blöcken ausgeführt.
- Auch reines `try-finally` (ohne `catch`) ist erlaubt.

finally: Ein Beispiel

```
try {
  try {
    System.out.println("eins");
    if(true) throw new Exception("hallo");
    System.out.println("zwei");
  } catch(MyException e) {
    System.out.println(e.getMessage());
  } finally {
    System.out.println("drei");
  }
} catch(Exception f) {
  System.out.println(f.getMessage());
} finally {
  System.out.println("vier");
}
```

finally: Ein Beispiel

```
try {
  try {
    System.out.println("eins");
    if(true) throw new MyException("hallo");
    System.out.println("zwei");
  } catch(MyException e) {
    System.out.println(e.getMessage());
  } finally {
    System.out.println("drei");
  }
} catch(Exception f) {
  System.out.println(f.getMessage());
} finally {
  System.out.println("vier");
}
```


Klasse Exception: Wichtige Methoden

- Konstruktor `Exception(String message)`: Gib der Exception eine Nachricht als String mit.
- `getMessage()`: hole die Nachricht wieder.
- `printStackTrace()`: Gib den Aufrufbaum aus, bei dem die Exception geworfen wurde.

Arbeiten mit Dateien

- Bisher haben wir Eingaben immer über die Kommandozeile eingelesen.
- In Java gibt es aber auch eine Reihe von Klassen, die mit Ein- und Ausgabe von Dateien zu tun haben.
- Zwei Arten von Klassen:
  - Byte-basiert: `InputStream`, `OutputStream`
  - Zeichen-basiert (Unicode): `Reader`, `Writer`

Dateien: Ein Beispiel

```
import java.io.*;

File f = new File("dateiname");

try {
  BufferedReader in =
    new BufferedReader(new FileReader(f));
  String line;

  do {
    line = in.readLine();
    System.out.println(line);
  } while( line != null );
} catch(IOException e) {
  e.printStackTrace();
}
```

Die Klasse File

- Klasse `File` repräsentiert einen Dateinamen.
- Repräsentation ist abstrahiert von Details der zugrundeliegenden Plattform (Windows/Unix).
- Wichtige Methoden:
  - Dateinamen aus Teilen zusammenbauen
  - Anlegen, Testen von Dateien
- Vor allem wichtig als Argument für Reader- und Stream-Konstruktoren.

Die Klasse Reader

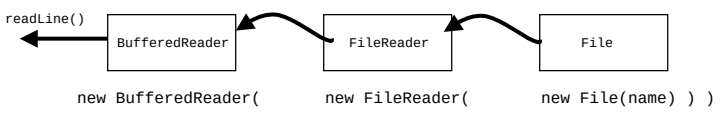
- Abstrakte Klasse zum Einlesen von zeichenbasierten Dateien.
- Automatische Übersetzung für Codierungen der Unicode-Zeichen, die Java intern nutzt.
- In der Praxis werden normalerweise abgeleitete konkrete Klassen verwendet.

Unterklassen von Reader

- `FileReader`: liest Zeichen aus einer Datei.
- `BufferedReader`: zusätzliche Methoden zum zeilenweisen Lesen.
- `StringReader`: liest Zeichen aus einem String.
- noch einige weitere

Verkettung von Readers

- Manche Reader haben einen Konstruktor, der einen anderen (beliebigen) Reader als Argument nimmt.
- Dieser Reader liest Zeichen vom anderen Reader und macht etwas Interessantes damit.



Die Klasse Writer

- Dual zu den Reader-Klassen gibt es Writer-Klassen, die zeichenbasiert schreiben.
- Einige Unterklassen:
  - `FileWriter`: schreibt in eine Datei
  - `PrintWriter`: zusätzliche Methoden zur Ausgabe verschiedener Datentypen (`print` und `println`)
  - `StringWriter`: schreibt in einen StringBuffer
- Verkettung genauso wichtig wie bei Readers

Streams

- Streams sind Klassen für byte-basierte Ein- und Ausgabe.
- InputStreams unterstützen Methode read(), die ein einzelnes Byte liest.
- OutputStreams unterstützen Methode write(), die ein einzelnes Byte schreibt.
- Wieder verschiedene konkrete Implementierungen (incl. für ganze Objekte!)
- Verkettungen

IOExceptions

- Die meisten Konstruktoren und Methoden, die mit Ein- und Ausgabe zu tun haben, werfen `IOException`s, wenn Fehler auftreten.
- `IOException`s sind Checked Exceptions, müssen also gefangen oder in `throws`-Klausel deklariert werden.

Zusammenfassung

- Ausnahmen (Exceptions) dienen zur kontrollierten Behandlung von Fehlern.
- Wenn Exception geworfen wird, setzt Verarbeitung dort fort, wo sie gefangen wird.
- Zwei Arten von Ein-/Ausgabeklassen:
  - Reader/Writer: zeichenbasiert
  - Streams: bytebasiert