CARL VON OSSIETZKY universität OLDENBURG

Programmierkurs Java

UE 21 - Exceptions

Dr.-Ing. Dietrich Boles

Gliederung

- Programmierfehler
- Rollen
- Motivation Fehlerbehandlung
- Traditionelle Fehlerbehandlung
- Fehlerbehandlung in Java im Überblick
- Fehlerklassen
- Fehlertypdeklaration
- "Werfen" von Fehlerobjekten
- "Abfangen" von Fehlerobjekten
- Weiterleiten von Fehlerobjekten
- Umgang mit Fehlerobjekten / Beispiel
- Beispiel Klasse IO
- Zusammenfassung

Programmierfehler

- Syntaxfehler
  - Fehlendes Semikolon
  - Zugriff auf nicht deklarierte Variable
  - → Compiler
- Logische Fehler
  - Ungewollte oder falsche Ergebnisse
  - → Testen
- Laufzeitfehler
  - Division durch 0
  - Zugriff auf nicht-existierendes Array-Element
  - Falsche Dateinamen
  - „Programmabsturz"
  - → Exceptions

Rollen

CARL VON OSSIETZKY universität OLDENBURG

- Nutzer
- Programmierer
  - API-Programmierer: definiert Klassen
  - Anwendungsprogrammierer: nutzt Klassen

Motivation Fehlerbehandlung

```
public class Mathematik { // API-Programmierer
  public static int fak(int n) {
    if (n > 0) return n * fak(n-1);
    else return 1;  // falsch, wenn n < 0 (nicht def.)
  }
  public static double div(double wert, double durch) {
    if (durch != 0.0) return wert/durch;
    else return 0.0;  // eigentlich falsch (unendlich)
} }

class Berechnungen { // Anwendungsprogrammierer
  public static void main(String[] args) {
    double eingabe = IO.readDouble();
    IO.println(Mathematik.fak(eingabe));
    IO.println(Mathematik.div(eingabe, eingabe-2.0));
} }
```

Ausgabe einer Fehlermeldung:

```
public class Mathematik { // API-Programmierer
  public static int fak(int n) {
    if (n > 0) return n * fak(n-1);
    else if (n == 0) return 1;
    System.err.println("ungueltiger Parameter");
    return -1;
} }
class Berechnungen { // Anwendungsprogrammierer
  public static void main(String[] args) {
    int eingabe = IO.readInt();
    IO.println(Mathematik.fak(eingabe));
} }
```

Problem: Anwendungsprogrammierer bekommt den Fehler nicht mit!

Benutzung einer error-Variablen:

```
public class Mathematik { // API-Programmierer
  public static final int NOERROR = 0;
  public static final int INVALID = 1;
  public static int error;
  public static int fak(int n) {
    if (n > 0)  { error = NOERROR; return n*fak(n-1); }
    if (n == 0) { error = NOERROR; return 1; }
    error = INVALID; return -1;
} }

    // Anwendungsprogrammierer
    int eingabe = IO.readInt();
    int result = Mathematik.fak(eingabe);
    if (Mathematik.error == Mathematik.NOERROR)
        IO.println(result);
    else if (Mathematik.error == Mathematik.INVALID)
        System.err.println("ungueltiger Parameter");
      ...
```

Problem: Ausgesprochen umständlich!

Fehlerbehandlung in Java / Überblick (1)

- Bei der Klassen-/Methodendefinition (API-Programmierer):
  - überlegen, was für Fehler auftreten können
  - Definition und Implementierung geeigneter "Fehlerklassen"
  - Erweiterung der Methodensignatur um Angabe möglicher Fehler
  - erzeugen und liefern ("werfen") von "Fehlerobjekten" im Falle eines Fehlereintritts

- Bei der Klassen-/Methodennutzung (Anwendungsprogrammierer):
  - beim Aufruf einer Methode ermitteln, welche Fehler prinzipiell auftreten können
  - zunächst "versuchen", die Methode auszuführen
  - falls kein Fehler auftritt, normal weitermachen
  - falls ein Fehler auftritt, das Fehlerobjekt abfangen und geeignete Fehlerbehandlung einleiten

Fehlerbehandlung in Java / Überblick (2)

- überlegen, was für Fehler auftreten können

```
public static int fak(int n) {
  if (n > 0)
    return n * fak(n-1);
  else if (n == 0)
    return 1;
  else
    ????????
}
```

Fehlerquelle: ungültiger aktueller Parameterwert (n < 0)

- Definition und Implementierung geeigneter "Fehlerklassen"

```
public class InvalidParameter
   extends Exception
{
      int actParamValue;
      public InvalidParameter(int value) {
         super("Ungültiger Parameter");
         this.actParamValue = value;
      }
      public int getParamValue() {
         return this.actParamValue;
      }
}
```

Fehlerbehandlung in Java / Überblick (4)

- Erweiterung der Methodensignatur um Angabe möglicher Fehler
- erzeugen und liefern ("werfen") von "Fehlerobjekten" im Falle eines Fehlereintritts

```
public static int fak(int n)
   throws InvalidParameter
{
   if (n > 0)
      return n * fak(n-1);
   else if (n == 0)
      return 1;
   else {
      InvalidParameter errorObj =
         new InvalidParameter(n);
      throw errorObj;
} }
```

Fehlerbehandlung in Java / Überblick (5)

CARL VON OSSIETZKY universität OLDENBURG

- beim Aufruf einer Methode ermitteln, welche Fehler prinzipiell auftreten können

```
static void doSomething() {
  int eingabe = IO.readInt();
  int result = fak(eingabe);
  for (int i=0; i<result; i++)
    IO.println(i);
}
```

möglicher Fehler: ungültiger aktueller Parameterwert für `fak`-Aufruf (`eingabe` < 0)

Fehlerbehandlung in Java / Überblick (6)

- zunächst "versuchen", die Methode auszuführen
- falls kein Fehler auftritt, normal weitermachen

```
static void doSomething() {
  try {
    int eingabe = IO.readInt();
    int result = fak(eingabe);
    for (int i=0; i<result; i++)
      IO.println(i);
  }
}
```

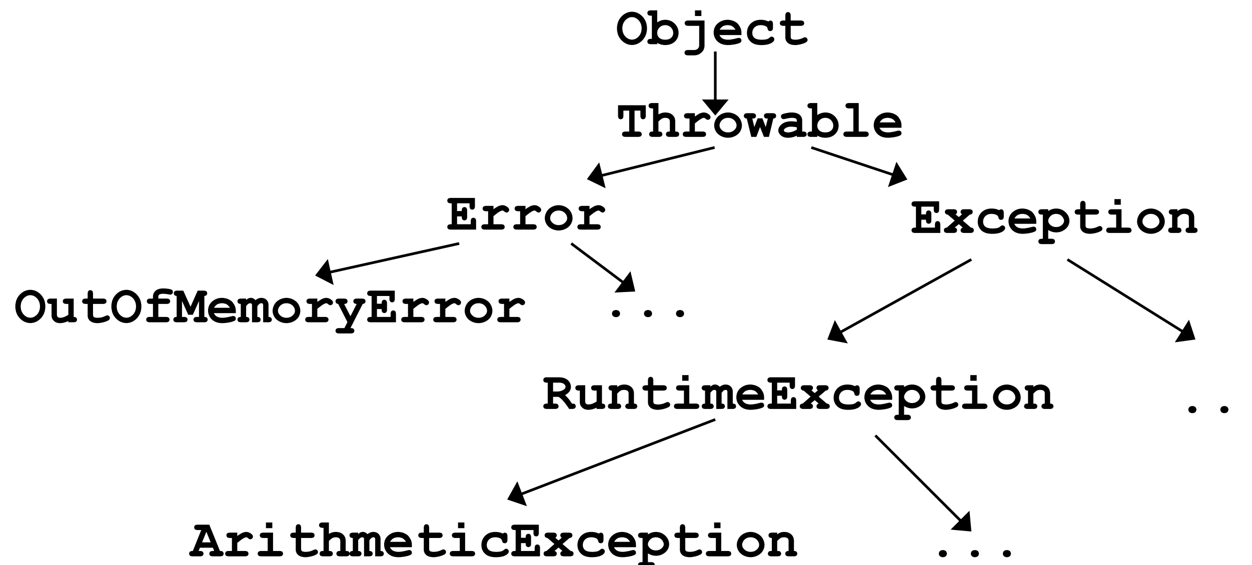
- **Motivation**: Trennung von Normalfall und möglichen Fehlerfällen

Fehlerbehandlung in Java / Überblick (7)

- falls ein Fehler auftritt, das Fehlerobjekt abfangen und geeignete Fehlerbehandlung einleiten

```
static void doSomething() {
  try {
    int eingabe = IO.readInt();
    int result = fak(eingabe);
    for (int i=0; i<result; i++)
      IO.println(i);
  } catch (InvalidParameter errorObj) {
    System.err.println("ungültige Eingabe: " +
       errorObj.getParamValue() + "wiederholen");
    doSomething();  // erneuter Versuch
} }
```

Fehlerklassen (1)



- **`Error`**: schwerwiegende Fehler
- **`RuntimeException`**: schwerwiegende Fehler
- Fehlerobjekte (*Exceptions*) sind Objekte von Fehlerklassen (von **`Throwable`** (zumeist indirekt) abgeleitete Klassen!)
- neue Fehlerklassen im Allgemeinen von **`Exception`** ableiten!
- Name einer Fehlerklasse sollte aussagekräftig sein!

Fehlerklassen (2)

```
package java.lang;

public class Exception extends Throwable {
  // Konstruktoren
  public Exception();
  public Exception(String message);
  // Methoden
  public String getMessage(); // liefert Beschreibung
  public String toString();   // ruft getMessage auf
}
```

Fehlertypdeklaration (1)

- Deklaration von Fehlertypen einer Methode in der Methodensignatur mittels des Schlüsselwortes **`throws`**
- Eine Methode **muss** alle Fehlertypen deklarieren, die sie werfen kann
- **`Error`**- und **`RuntimeException`**-Fehler müssen nicht unbedingt deklariert werden (Unchecked-Exceptions)

```
public class Ex1 extends Exception {}
public class Ex2 extends Exception {}

public int f() throws Ex1, Ex2 {
   ... if (...) throw new Ex1(); ...
   ... if (...) throw new Ex2(); ...
}
```

Fehlertypdeklaration (2)

- Die Fehlertypdeklaration gehört mit zur Signatur einer Methode
- Werden Methoden mit Fehlertypen in abgeleiteten Klassen **überschrieben**, dann müssen die Fehlertypdeklarationen übereinstimmen oder es müssen Fehlerklassen angegeben werden, die von den Fehlerklassen der überschriebenen Methode abgeleitet sind!

```
public class Ex1 extends Exception {}
public class Ex2 extends Exception {}
public class Ex3 extends Ex2 {}
public class A {
  public int f(float v) throws Ex1, Ex2 { ... }
}
public class B extends A { // nur eine der Alternativen
  public int f(float v) throws Ex1, Ex2 { ... }
  public int f(float v) throws Ex1, Ex3 { ... }
}
```

"Werfen" von Fehlerobjekten

- "Werfen" von Fehlerobjekten mittels des Schlüsselwortes **`throw`**
- nach dem Werfen eines Fehlerobjektes wird eine Methode (wie beim **`return`**) direkt verlassen
- Es können nur Objekte solcher Fehlerklassen geworfen werden, die auch in der Methodendeklaration aufgeführt sind

```
public class InvalidParam extends Exception {}

public class Math {
  public static int fak(int n) throws InvalidParam {
    if (n < 0) throw new InvalidParam();
    if (n == 0) return 1;
    return n * fak(n-1);
} }
```

"Abfangen" von Fehlerobjekten / try-Block

- Methoden, die (andere) Methoden aufrufen, die Fehlerobjekte werfen können, müssen diese abfangen und behandeln oder weiterleiten
- Zum Abfangen eines Fehlerobjektes muss die entsprechende Methode innerhalb eines so genannten **try-Blockes** aufgerufen werden
- Die Fehlerbehandlung wird in einem so genannten
  "den Fehlertyp matchenden"-catch-Block durchgeführt
- wird ein Fehlerobjekt geworfen, so wird die Bearbeitung des try-Blockes unmittelbar beendet
- existiert ein matchender-catch-Block, werden die Anweisungen dieses Blockes ausgeführt

```
try {
  ...
} catch (XYException obj) { /* Fehlerbehandlung */ }
```

"Abfangen" von Fehlerobjekten / catch-Block

- Sehr große Ähnlichkeiten zu Prozeduren und Parametern
- muss einem try-Block (oder einem anderen catch-Block) folgen
- "formale Parameter" müssen Fehlerklassen/-Objektvariablen sein, die im try-Block auftreten können (bzw. Oberklassen der Fehlerklassen (→ Polymorphie))
- die Fehlerobjektübergabe ist identisch zur Parameterübergabe bei Funktionen, d.h. insbesondere können die formalen Fehlerobjekte wie lokale Variablen des catch-Blockes behandelt werden; sie werden mit dem geworfenen Fehlerobjekt initialisiert
- wird während der Ausführung des try-Blockes ein Fehlerobjekt geworfen, so wird der try-Block verlassen, und es wird der Reihe nach (!) überprüft, ob ein catch-Block mit dem Fehlerobjekt *matched*; dabei gilt:
- **ein catch-Block *matched*** ein aktuelles Fehlerobjekt, wenn die Klasse seines formalen Fehlerobjektes gleich der Klasse oder eine Oberklasse des aktuellen Fehlerobjektes ist

"Abfangen" von Fehlerobjekten / finally-Block (1)

- existiert ein so genannter **finally-Block**, wird dieser auf jeden Fall ausgeführt

```
try {
   ...
} catch (XException obj) {
   // Fehlerbehandlung X-Fehler
} catch (YException obj) {
   // Fehlerbehandlung Y-Fehler
} finally {
   // wird IMMER ausgeführt!
}
```

"Abfangen" von Fehlerobjekten / finally-Block (2)

- ist "parameterlos"
- schließt einen try-{catch}*-Block ab, d.h. kann hinter einem bzw. einer Menge von catch-Blöcken definiert werden
- ist ein finally-Block hinter einem try-{catch}*-Block vorhanden, so wird dieser auf jeden Fall ausgeführt, d.h.
  - wenn kein Fehlerobjekt im try-Block geworfen wurde
  - wenn ein Fehlerobjekt im try-Block geworfen und nicht abgefangen wurde (nach dem finally-Block wird in diesem Fall die Funktion verlassen und das Fehlerobjekt an die aufrufenden Funktion weitergeleitet)
  - wenn ein Fehlerobjekt im try-Block geworfen und abgefangen wurde, d.h. nach der Ausführung des matchenden catch-Blockes

Weiterleiten von Fehlerobjekten

- Fehlerobjekte können auch weitergeleitet werden

```
public class Ex1 extends Exception {}
public class Ex2 extends Exception {}

public int f() throws Ex1, Ex2 {
   ... if (...) throw new Ex1(); ...
   ... if (...) throw new Ex2(); ...
}

public int g() throws Ex1 {
   try { ... int i = f(); ... } catch (Ex2 e) {...}
}
// Fehler Ex2 wird behandelt
// Fehler Ex1 wird weitergeleitet
```

Umgang mit Fehlerobjekten / Beispiel (1)


```
public int f(double v) throws Ex1, Ex2, Ex3 { ... }
public float g(int i) throws Ex1, Ex3, Ex4 { ... }
public void h() throws Ex3 {
  ... (*1)
  try {
     ... (*2)
     int r = f(2.0);
     ... (*3)
     float x = g(r);
     ... (*4)
  }
  catch (Ex1 errorEx1) { ... (*5) }
  catch (Ex2 errorEx2) { ... (*6) }
  catch (Ex4 errorEx4) { ... (*7) }
  finally { ... (*8) }
  ... (*9)
}
```

Umgang mit Fehlerobjekten / Beispiel (2)

CARL VON OSSIETZKY universität OLDENBURG

Erläuterungen (zum Beispiel vorher):

1. wird die Funktion `h` (bspw. von einer Funktion `s`) aufgerufen, dann wird zunächst `(*1)` ausführt
2. anschließend wird `(*2)` ausgeführt
3. wird während der Ausführung der Funktion `f` kein Fehlerobjekt geworfen, dann wird `(*3)` ausgeführt (weiter bei 6.)
4. wird während der Ausführung der Funktion `f` ein Fehlerobjekt vom Typ `Ex3` geworfen, dann wird zunächst `(*8)` ausgeführt und danach die Funktion `h` unmittelbar verlassen und das Fehlerobjekt an die aufrufende Funktion `s` weitergeleitet
5. wird während der Ausführung der Funktion `f` ein Fehlerobjekt vom Typ `Ex1` (oder `Ex2`) geworfen, dann wird dieses Objekt (wie bei der Parameterübergabe) als aktueller Parameter dem formalen Parameter `errorEx1` (bzw. `errorEx2`) übergeben und `(*5)` (bzw. `(*6)`) ausgeführt (weiter bei 10.)

Umgang mit Fehlerobjekten / Beispiel (3)

CARL VON OSSIETZKY universität OLDENBURG

6. nach der Ausführung von `(*3)` wird die Funktion `g` aufgerufen
7. wird während der Ausführung der Funktion `g` kein Fehlerobjekt geworfen, dann wird `(*4)` ausgeführt (weiter bei 10.)
8. wird während der Ausführung der Funktion `g` ein Fehlerobjekt vom Typ `Ex3` geworfen, dann wird zunächst `(*8)` ausgeführt und danach die Funktion `h` unmittelbar verlassen und das Fehlerobjekt an die aufrufende Funktion `s` weitergeleitet
9. wird während der Ausführung der Funktion `g` ein Fehlerobjekt vom Typ `Ex1` (oder `Ex4`) geworfen, dann wird dieses Objekt (wie bei der Parameterübergabe) als aktueller Parameter dem formalen Parameter `errorEx1` (bzw. `errorEx4`) übergeben und `(*5)` (bzw. `(*7)`) ausgeführt
10. nach Ausführung von `(*4)` , `(*5)` , `(*6)` oder `(*7)` wird auf jeden Fall `(*8)` ausgeführt
11. anschließend wird `(*9)` ausgeführt und die Funktion beendet

Beispiel Klasse IO (1)

```
class IOException extends Exception

class BufferedReader { // java.io
  String readLine() throws IOException
}

class NumberFormatException extends RuntimeException

class Integer { // java.lang
  Integer(String s) throws NumberFormatException
}
```

Beispiel Klasse IO (2)

```
class IO {
    public static int readInt() {
        try {
            BufferedReader input =
              new BufferedReader(
                new InputStreamReader(System.in));
            String eingabe = input.readLine();
            return new Integer(eingabe);
        } catch (Throwable exc) {
            return 0;
        }
    }
}
```

Zusammenfassung

- Exceptions dienen zur Behandlung von potentiellen Laufzeitfehlern
- Motivation: Trennung von "Normalfall" und möglichen Fehlerfällen
- Exceptions sind Objekte von von der Klasse `Throwable` abgeleiteten Klassen
- Im Fehlerfall wird von einer Methode eine Exception geworfen
- aufrufende Methoden können diese Exception abfangen und eine Fehlerbehandlung einleiten

- Sinnvoller Einsatz von Exceptions: ADT-Klassen
- nicht sinnvoll: Ersetzen von "normalen" if-Anweisungen zur Programmsteuerung