gds2 Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

Inhaltsverzeichnis

1 Grundlagen

„Exceptions“ sind Fehler, die während der Programmausführung, also zur Laufzeit auftreten. Tritt eine Exception auf, so kann diese abgefangen oder weitergeleitet werden ("catch or throw"-Regel).

Eine Exception ist technisch gesehen ein Objekt der Klasse `Exception`. Um die verschiedenen Fehlermöglichkeiten individuell behandeln zu können werden von der Basisklasse `Exception` viele Unterklassen abgeleitet – für jede Fehlerart eine:

Direkte Unterklassen der Klasse `Exception` sind:

```
AclNotFoundException, ActivationException, AlreadyBoundException,
ApplicationException, AWTException, BackingStoreException,
BadLocationException, CertificateException, ClassNotFoundException,
CloneNotSupportedException, DataFormatException, DestroyFailedException,
ExpandVetoException, FontFormatException, GeneralSecurityException,
GSSException, IllegalAccessException, InstantiationException,
InterruptedException, IntrospectionException, InvalidMidiDataException,
InvalidPreferencesFormatException, InvocationTargetException, IOException,
LastOwnerException, LineUnavailableException, MidiUnavailableException,
MimeTypeParseException, NamingException, NoninvertibleTransformException,
NoSuchFieldException, NoSuchMethodException, NotBoundException,
NotOwnerException, ParseException, ParserConfigurationException,
PrinterException, PrintException, PrivilegedActionException,
PropertyVetoException, RefreshFailedException, RemarshalException,
RuntimeException, SAXException, ServerNotActiveException, SQLException,
TooManyListenersException, TransformerException,
UnsupportedAudioFileException, UnsupportedCallbackException,
UnsupportedFlavorException, UnsupportedLookAndFeelException,
URISyntaxException, UserException, XAException
```

Jede dieser Klassen kann weitere Unterklassen enthalten.

Man unterscheidet dabei kontrollierte und unkontrollierte Exceptions:
Kontrollierte Exceptions müssen abgefangen oder weitergeleitet werden, sonst gibt es einen Compiler-Fehler. Unkontrollierte Exceptions müssen nicht behandelt werden. Unkontrollierte Exceptions werden alle von der Klasse `RuntimeException` abgeleitet.

2 Entstehung von Exceptions

Exceptions werden von Methoden erzeugt. Man sagt auch: eine Exceptions wird "geworfen". Welche Exception von welcher Methode "geworfen" wird, lässt sich der zugehörigen API Dokumentation entnehmen.

So wirft z.B. die Methode `parseInt` aus der Klasse `Integer` eine `NumberFormatException`, wenn der umzuwandelnde `String s` keinen umwandelbaren `int` enthält:

```
public static int parseInt(String s) throws NumberFormatException
```

Diese Exception muss nicht abgefangen werden, da sie von der Klasse `RuntimeException` abgeleitet ist:

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └java.lang.IllegalArgumentException
                  └java.lang.NumberFormatException
```

3 Behandlung von Exceptions

Gemäß der "Catch-or-throw"-Regel können Exceptions entweder abgefangen oder weitergeleitet werden.

3.1 Weiterleiten von Exceptions

Hinter dem Methodennamen werden mit dem Schlüsselwort `throws` die weitergeleiteten Exceptions aufgelistet, ggf. durch Komma getrennt:

Beispiel:

```
public int toInt (String sZahl) throws NumberFormatException
{
  return (Integer.parseInt(sZahl));
}
```

3.2 Abfangen von Exceptions

3.2.1 try - catch

Soll eine Exception nicht weitergeleitet, sondern abgefangen und direkt behandelt werden, so muss die Methode, die die Exception auslöst, in einen `try-catch`-Block eingeschlossen werden.

Beispiel:

```
public void toInt (String sZahl)
{
  int iZahl;
  try
  {
    iZahl = Integer.parseInt(sZahl);
    System.out.println("umgewandelte Zahl:"+iZahl);
  }
  catch (NumberFormatException nfe)
  {
    System.out.println(sZahl+" kann nicht umgewandelt werden");
  }
}
```

Tritt in diesem Beispiel ein Umwandlungsfehler auf, so wird der try-Block sofort verlassen und der catch-Block mit der passenden Exception angesprungen.
Es können auch mehrere catch-Blöcke hintereinander folgen. Hierbei ist allerdings zu beachten,

dass diese in Richtung der Basisklasse angeordnet werden. Ein catch-Block, der die Klasse `Exception` abfängt, sollte also als letztes geschrieben werden, da ansonsten ein `catch`-Block einer abgeleiteten Klasse nie angesprungen würde.

3.2.2 finally

Optional kann nach den catch-Blöcken ein `finally`-Block folgen. Dieser wird immer betreten, wenn

- der zugehörige try-Block normal beendet wurde
- eine durch einen catch-Block abgefangene Exception aufgetreten ist
- eine durch einen catch-Block nicht abgefangene Exception aufgetreten ist
- der try-Block durch break, continue oder return verlassen werden soll

Hier können z.B. "Aufräumarbeiten" wie das Schließen von Dateien vorgenommen werden.

Beispiel:

```
public static void main(String[] args)
{
  BufferedReader f = null;
  Path dateiPfad = Paths.get("Dateien/Lesen.txt");
  String zeile;
  try
  {

    f = Files.newBufferedReader(dateiPfad, StandardCharsets.UTF_8);
    zeile = f.readLine();
    System.out.println(zeile);
  }
  catch (IOException e)
  {
    System.out.println("Fehler: " + e.getMessage());
  }
  finally
  {
    if (f != null)
    {
      try
      {
        f.close();
      }
      catch (IOException ign)
      {
      }
    }
  }
}
```

3.2.3 "try with resources"

Im obigen Beispiel ist unschwer zu erkennen, dass das Schließen einer Ressource relativ viel Code erfordert, insbesondere wenn `close()` korrekt behandelt werden soll, nämlich incl. Abfangen einer `IOException`.

Seit Java SE 7 gibt dafür eine elegante Möglichkeit in Form einer erweiterten `try`-Anweisung: "try with resources": alle hinter `try` in runden Klammern aufgeführten Ressourcen werden automatisch geschlossen, sobald der `try`-Block abgearbeitet wurde, unabhängig davon, ob eine Exception erzeugt wurde oder nicht.

```
public static void main(String[] args)
{
  BufferedReader f = null;
  Path dateiPfad = Paths.get("Dateien/Lesen.txt");
  String zeile;
  try (f = Files.newBufferedReader(dateiPfad, StandardCharsets.UTF_8))
  {

    ;
    zeile = f.readLine();
    System.out.println(zeile);
  }
  catch (IOException e)
  {
    System.out.println("Fehler: " + e.getMessage());
  }
}
```

Um diesen Mechanismus zu ermöglichen wurde mit Java 7 das Interface `java.lang.AutoCloseable` eingeführt. Somit kann "try with resources" auch für eigene Klassen genutzt werden, wenn sie dieses Interface implementieren.

4 Eigene Exceptions auslösen

Auch in eigenen Methode kann man Exceptions auslösen. Damit ist es möglich, auch in selbst geschriebenem Code dasselbe Prinzip der Fehlerbehandlung einzusetzen wie das JDK.

Vorgehensweise:
Um eigene Exceptions auszulösen geht man in 2 Schritten vor:

1. ein neues Objekt einer bereits vorhandenen Exception-Klasse erzeugen
2. dieses Objekt mit `throw` "werfen", d.h. die Exception auslösen

Beispiel:

```
   public class ExceptionTest
   {
     public static void main(final String[] args)
     {
       try
       {
         vPruefeWert (999);
         System.out.println("Wert ok");
       }
       catch (IndexOutOfBoundsException e)
       {//"Fehler:Ungültiger Wert"
1        System.out.println("Fehler:"+e.getMessage());
       }
     }

2    public static void vPruefeWert (int iWert) throws
     IndexOutOfBoundsException
     {
       if (iWert < 1 || iWert > 100)
       {
3        throw new IndexOutOfBoundsException ("Ungültiger Wert");
       }
     }
   }
```

Anmerkungen :

2	Die Methode wird – wie üblich – gekennzeichnet
3	Ein Objekt der Klasse `IndexOutOfBoundsException` wird erzeugt, indem ihr Konstruktor aufgerufen wird (auch das ist nichts Neues).
1	Beachtenswert ist die Tatsache, dass dem Konstruktor ein beliebiger Text übergeben werden kann, der dann über die Methode `getMessage()`, die von der Klasse `Exception` geerbt wurde, ausgegeben werden kann.

5 Eigene Exception-Klassen erstellen

Um eine eigene Exception-Klasse zu erstellen muss nur eine der vorhandenen Exception-Klassen erweitert werden.
Die neue Exception-Klasse sollte mindestens 2 Konstruktoren haben – einen ohne Parameter und einen mit einem Meldungstext. Sie kann um beliebige eigene Methoden zur Fehleranalyse ergänzt werden.

Beispiel:

```
1  public class WertUngueltigException extends Exception
   {
2    public WertUngueltigException ()
     {
       super();
     }
3    public WertUngueltigException (String sFehlertext)
     {
       super(sFehlertext);
     }
4    public WertUngueltigException (int iMin, int iMax)
     {
5      super("Wert liegt nicht zwischen "+iMin + " und " +iMax);
     }
   }
```

```
   public class EigeneExceptionTest
   {
     public static void main(final String[] args)
     {
       try
       {
         vPruefeWert (999);
         System.out.println("Wert ok");
       }
       catch (WertUngueltigException e)
       {
         System.out.println("Fehler:"+e.getMessage());
         //Ausgabe "Fehler: Wert liegt nicht zwischen 1 und 100"
       }
     }

     public static void vPruefeWert (int iWert) throws
   WertUngueltigException
     {
       if (iWert < 1 || iWert > 100)
       {
6        throw new WertUngueltigException (1, 100);
       }
     }
   }
```

Anmerkungen :

1	Erstellen der neuen Exception-Klasse (sie könnte auch von einer Unterklasse von `Exception` abgeleitet werden)
2,3,4	3 Konstruktoren der Klasse `WertUngueltigException`, die zum Erzeugen eines Objekts benutzt werden können ...
6	... in diesem Beispiel wird der Konstruktor Nr. 4 benutzt
5	super: mit diesem Schlüsselwort wird der Konstruktor der jeweiligen Oberklasse aufgerufen, in diesem Fall also der Konstruktor der Klasse `Exception`