1.1 Ausnahme (Exception)

1.1.1 Motivation

1.1.1.1 Umgang mit Fehlern im Alltag

Im Englischunterricht in der Schule werden viele Fehler gemacht (Wörter falsch sprechen oder schreiben), in Firmen werden oft Fehler gemacht (z.B. Produktionsfehler), überall werden Fehler gemacht ...
Damit diese Fehler keine Auswirkungen haben (z.B. in die Hände des Kunden fallen), werden sie einer internen Kontrolle unterworfen (Qualitätssicherung?) und dann selbst gleich beseitigt, oder es werden die Fehler an eine übergeordnete Instanz weitergeleitet, die sie dann selbst bearbeitet oder wieder weiterleitet ...

1.1.1.2 Umgang mit Fehlern in Java

Zur Laufzeit eines Javaprogramms können bei der Abarbeitung einer Anweisung sogenannte Ausnahmen (Exception) eintreten - das sind außerordentliche - durch Fehler verursachte - Bedingungen, wie z.B. die Division durch Null oder der Zugriff auf ein sich außerhalb eines Feldes befindlichen Elements - die es nicht gestatten das Programm weiterhin "normal" abzuarbeiten.
In derartigen Fällen wird automatisch in der Methode, in der die Ausnahme aufgetreten ist, ein sogenanntes Fehlerobjekt (der Klasse Throwable oder einer Unterklasse davon) erzeugt und ausgeworfen.

Bemerkung:
1) Nur ganze Zahlen wie z.B. 16 bewirken bei einer Division durch 0 (wie z.B. 16 / 0) die Erzeugung eines Fehlerobjekts. Bei float- oder double-Zahlen bewirkt dies keine Ausnahme, sondern liefert einen speziellen Wert ("unendlich"). Für die Abfrage dieser Werte gibt es in den Klassen Float und Double entsprechende Konstanten und Methoden. Bitte in einem Programm ausprobieren: 16.0/0 bzw. 16/0

1.1.1.3 Wer erzeugt Fehlerobjekte und wirft sie aus ?

Die Java Virtual Machine kann Fehlerobjekte erzeugen und auswerfen.
Aber genauso hat der Programmierer die Möglichkeit, eigene Fehlerobjekte zu erzeugen und diese auszuwerfen.

1.1.2 Wann Fehler einfangen

Für manche (aber nicht alle) Anweisungen verlangt die Java Language Specification (JLS), dass die im Fehlerfall geworfenen Fehlerobjekte "eingefangen" (abgefangen und bearbeitet) werden (programmtechnisch realisiert mit **try ... catch**), oder das Fehlerobjekt an die umgebende Methode weitergeworfen wird (programmtechnisch realisiert durch den Bezeichner "**throws**" im Methodenkopf der umgebenden Methode).
Man sagt auch: "Problembereiche einzäunen". Näheres dazu später.

Gemäß der Java Language Specification (JLS) gilt:
1) Fehler, die Fehlerobjekte vom Klassentyp Error oder RuntimeException (oder deren Unterklassen) werfen, müssen (können aber) nicht vom Programmierer "eingefangen" bzw. "weitergeworfen" werden. Man nennt sie ungeprüfte Ausnahmen (unchecked Exceptions).
Die Fehlerobjekte der - siehe unten - **fett dargestellten** Klassen (und deren Unterklassen) **müssen** also **nicht** vom Programmierer **eingefangen** werden.

2) Die Fehlerobjekte der - siehe unten - **NICHT fett dargestellten** Klassen (und deren Unterklassen) **MÜSSEN** vom Programmierer **eingefangen** werden.

Komplizierter ausgerückt:
Alle Fehlerobjekte der Klasse "Throwable" und deren Unterklassen (außer der Unterklasse RuntimeException bzw. Unterklassen davon und der Klasse Error bzw. Unterklassen davon) müssen vom Programmierer "eingefangen" bzw. "weitergeworfen" werden .

1.1.2.1 Bemerkung

Die Eingabe von Daten über Tastatur wird in Java auch über Exceptions realisiert.

1.1.3 Wie Fehler einfangen

Mit der try {...}... catch {...} Anweisung kann man Fehlerobjekte wieder einfangen.
Diese Anweisung besteht aus einem try-Bereich und dem catch-Bereich, der aus mindestens einem catch-Block besteht.

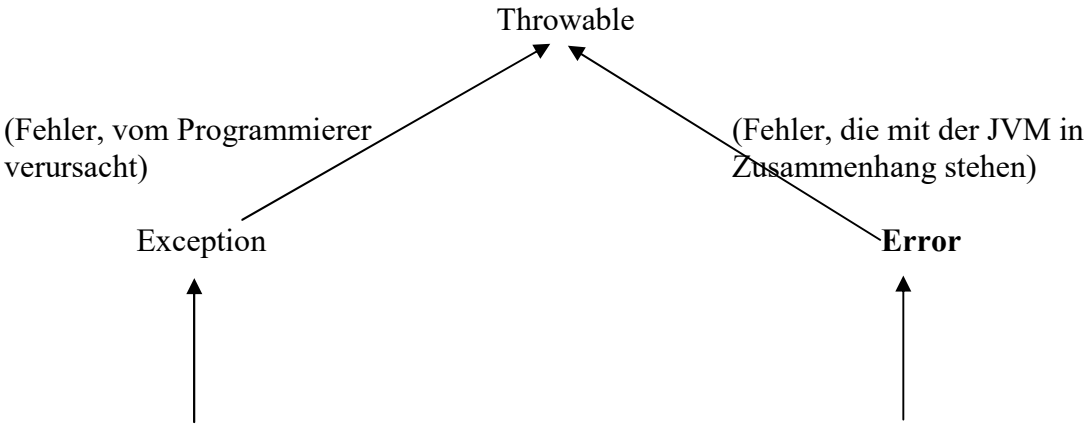
```
try{
  Anweisung(en);
}
catch(Throwable t){
  Anweisung(en);
}
finally{
  Anweisung(en);
}
```

Die (kritische) Anweisung, die einen Fehler verursachen kann, schreibt man in den try-Bereich der try {...}... catch {...} Anweisung. Die catch-Bezeichner werden auch Ausnahme-Handler genannt. Es muss **mindestens** einen catch-Bezeichner geben, der dann im Fehlerfall abgearbeitet wird (Ausnahme: Es gibt ein try ohne catch aber dann mit finally; näheres weiter unten).
Der formale Parameter des catch-Bezeichners (hier mit t bezeichnet) muss als Klassentyp "Throwable" oder eine Unterklasse davon haben.
Der finally-Teil ist optional, kann also weggelassen werden. Falls aber finally benutzt wird, werden die in im finally-Teil stehenden Anweisung auf jeden Fall abgearbeitet (egal, ob ein Fehlerobjekt geworfen wird oder nicht).

1.1.4 Ausnahmen und Klassen

Die sich in der Vererbungshierarchie am weitesten oben befindliche Klasse ist "Throwable". darunter befinden sich die Unterklassen.

1.1.4.1 Schaubild Klassen Hierarchie



ClassNotFoundException
IOException
ClassNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchFieldException
NoSuchMethodException
RuntimeException

LinkageError
VirtualMachineError
ThreadDeath

1.1.4.1.1 Unterklassen von LinkageError

AbstractMethodError: Aufruf einer abstrakten Methode
NoSuchFieldError: Zugriff auf eine nicht deklarierte Variable
NoSuchMethodError: Zugriff auf eine nicht deklarierte Methode

1.1.4.1.2 Unterklassen von VirtualMachineError:

OutOfMemoryError, StackOverFlowError, InternalError, UnknownError

1.1.4.1.3 Unterklassen von RuntimeException:

ArithmeticException
NullPointerException
NumberFormatException: Zahlen liegen in einem falschen Format vor, wenn z.B. "abc" in den Datentyp int konvertiert werden soll.
ArrayIndexOutOfBoundsException: Zugriff mit zu großem oder kleinen Index auf ein Feldelement

Bemerkung:
Eine Unterklasse der Klasse IOException ist die Klasse FileNotFoundException. Von dieser wird ein Objekt geworfen, wenn eine Datei, auf die zugegriffen wird, nicht existiert.
Eine Unterklasse der Klasse IOException ist die Klasse EOFException. Von dieser wird ein Objekt geworfen, wenn der Dateizeiger über das Dateiende hinaus verschoben wurde und dann noch von dieser Datei gelesen werden soll.

1.1.5 Beispiel (geprüfte Ausnahme: Compiler bringt Fehler)

```
public class MainException1 {
  public static void main(String[] args) {
    // Diese Anweisung verursacht beim Compilieren
    // eine Fehlermeldung.
    Class myC = Class.forName("Affe");
    String myS = myC.getName();
    System.out.println("myS="+myS);
  }
}
```

Bemerkungen:
1) Die Methode forName(...) der Klasse Class "lädt" eine Klasse mit dem angegebenen Namen.
Im Beispiel unten soll eine Klasse mit dem Namen Affe geladen werden.

2) Die Methode getName() der Klasse Class liefert den Namen der Klasse

3) Die Methode forName wirft eine Exception vom Klassentyp ClassNotFoundException aus und muss laut obigem Schaubild "Klassenhierarchie" eingefangen werden.
Deshalb gibt es beim Kompilieren dieses Programms eine Fehlermeldung des Kompilers.

1.1.6 Beispiel (obiges, verbessertes Beispiel)

```
public class MainException2 {
  public static void main(String[] args) {
    try{
      Class myC = Class.forName("Affe");
      String myS = myC.getName();
      System.out.println("myS="+myS);
    }
    catch(Throwable t){
      System.out.println("Klasse existiert nicht");
      System.out.println("Klasse inexistent:" +t.toString());
      t.printStackTrace();
    }
  }
}
```

Bemerkungen:
1) Der Programmierer fängt durch eine try ... catch Anweisung das Fehlerobjekt ein.
2) Die Methode toString() liefert eine Zeichenkette, in der Infos über das Objekt stehen.
3) Die Methode printStackTrace() ruft intern zuerst die Methode toString auf und bringt dann noch die Zeilen des Quellcodes, wo sich ein Fehler ereignet hat.

Alternativ kann das letzte Beispiel auch so programmiert werden:

1.1.7 Beispiel (Alternative zum letzten Beispiel)

```
public class MainException3 {
  public static void main(String[] args) throws Throwable {
    Class myC = Class.forName("Affe");
    String myS = myC.getName();
    System.out.println("myS="+myS);
    }
}
```

Bemerkungen:
1) In der die Anweisung Class.forName("Affe") umgebenden Methode (hier also main) wird direkt nach dem Bezeichner "throws" die Klasse des Fehlerobjekts angegeben, das im Fehlerfall geworfen wird. In dem Beispiel ist das die Klasse Throwable.
Wenn das Fehlerobjekt von main (durch throws) weitergeworfen wird, wird von der JVM automatisch der sogenannte **Standard-Ausnahmehandler** aufgerufen und dieses Fehlerobjekt eingefangen und bearbeitet.
Es muß also kein try ... catch (hier in der Methode main(...)) verwendet werden, da ein eventuell auftretendes Fehlerobjekt an die umgebende Methode (hier also main) weitergeleitet wird.

1.1.8 Beispiel (ungeprüfte Ausnahme)

```
public class MainException4 {
  public static void main(String[] args) {
    double z;
    z = 15/0;
  }
}
```

Bemerkungen:
1) Die Anweisung `z=15/0` wirft ein Fehlerobjekt vom Klassentyp ArithmeticException (Unterklasse von RuntimeException) aus und muss (siehe oben) nicht eingefangen werden. Deshalb gibt es beim Kompilieren dieses Programms keine Fehlermeldung des Kompilers. Direkt nach der Erzeugung des Fehlerobjekts, wird der zu main gehörige Standard-Ausnahmehandler der JVM aufgerufen und das Programm beendet.

1.1.9 Genauere Beschreibung der Fehlerbehandlung

Voraussetzung:
Eine Methode wirft im Fehlerfall ein Fehlerobjekt aus.
Der Programmierer ruft diese Methode auf und es wird ein Fehlerobjekt geworfen.

Möglichkeiten des Programmierers:
1. Möglichkeit:
Wenn das Fehlerobjekt eingefangen werden muss, muss der Programmierer try ... catch oder throws verwenden.

2. Möglichkeit:
Wenn das Fehlerobjekt nicht eingefangen werden muss, kann der Programmierer try ... catch oder throws verwenden oder das Fehlerobjekt einfach ignorieren.
Falls der Programmierer das Fehlerobjekt ignoriert (keinen Ausnahme-Handler programmiert hat), wird von der JVM sofort nach Erzeugung des Fehlerobjekts automatisch der standardmäßig zu main gehörige Standard-Ausnahmehandler aufgerufen, der den Typ der Ausnahme und die Methodenaufrufe, die zur Ausnahme geführt haben, ausgibt.
Danach wird dann das Programm sofort beendet.

Im Folgenden behandeln wir die Möglichkeiten try ... catch oder throws:

1.1.9.1 Fall: Programmierer benutzt throws

Da sich dieser (wie jeder) Methodenaufruf innerhalb einer umgebenden Methode befinden muss (z.B. in main), muss im Methodenkopf der umgebenden Methode mit dem Bezeichner "throws" angegeben werden, welcher Fehlerklasse das ausgeworfene Fehlerobjekt angehört. Man hat das Problem also eine Ebene weiter zum Methodenaufruf der umgebenden Methode verschoben.
Wenn man dort auch nicht try ... catch benutzen will, muss man wieder mit "throws" das Fehlerobjekt an die umgebende Methode weiterwerfen.

1.1.9.1.1 Beispiel 1

```
public class MainException5{
  public static void main(String[] args){
    double erg;
    try{
      erg=myDivision(15, 0);
    }
    catch(Throwable t){
      System.out.println("Division durch 0 unmöglich");

    }
  }

  public static double myDivision(int a, int b) throws
                                    Throwable{
    // muß nicht mit try...catch einezäunt werden
    double erg=a/b;
    return erg;
  }
}
```

Bemerkungen:
1) Da man (zu Demozwecken) in der Methode myDivision(...) nicht mit try ... catch das Fehlerobjekt einfangen will, hat man es einfach durch den Bezeichner throws in die umgebende Methode (dies ist hier main) weitergeleitet. Dort wird es dann mit try ... catch eingefangen. Falls man es dort nicht mit try ... catch einfangen will, kann man es wieder weiterleiten, indem man der Methode main den Bezeichner throws folgen lässt.

2) Da die Division durch 0 ein Fehlerobjekt erzeugt, das durch den Programmierer nicht eingefangen werden muss, würde man im obigen Beispiel kein try ... catch oder throws benötigen.

3) try ... catch ohne catch aber dann mit finally
setzt voraus, daß ein auftretendes Fehlerobjekt an die umgebende Methode weitergeleitet wird, aber vorher noch die Anweisungen des finally-Teils, abgearbeitet werden.

```
public void testMethode() throws Throwable{
  try{
    Anweisung(en);
  }
  finally{
    Anweisung(en);
  }
}
```

1.1.9.1.2 Beispiel 2

```
public class MainException5a{
  // Angabe von throws in main unnötig, da die hier in main
  // durch try .. catch das Fehlerobjekt eingefangen wird.
  // public static void main(String[] args) throws Throwable {
  public static void main(String[] args){
    double erg;
    try{
      erg=myTaschenrechner(5, 3, '/');
    }
    catch(Throwable t){
      System.out.println("Meine Fehlermeldung:");
      // mit getMessage() wird der String aus t geholt, der in der
      // Methode myTaschenrechner im Fehlerobjekt gesetzt wurde.
      System.out.println(t.getMessage());
      // auch möglich wie oben: toString() bzw. printStackTrace()
    }
  }

  public static double myTaschenrechner(int a, int b, char
                              rechenzeichen) throws Throwable{
    double erg;
    Throwable myt;

    if(rechenzeichen=='+'){
      erg=a+b;
    }
    else if(rechenzeichen=='-'){
      erg=a-b;
    }
    else{
      // Der durch myt = Throwable(...) festgelegte Text kann
      // mit getMessage() wieder gelesen werden (siehe main())
      myt = new Throwable("Taschenrechner kann nur + und -");
      // auch möglich, falls die Zeile oberhalb fehlt:
      // throw new Throwable("Mein Rechner kann nur + und -");
      // mit throw wird ein Fehlerobjekt geworfen.
      throw myt;
      // Das geworfene Fehlerobjekt myt wird oben im
      // Methodenkopf mit throws an die die umgebende Methode
      // weiter geworfen; return erg wird nicht mehr gemacht!
    }
    return erg;
  }
}
```

Bemerkungen:
Dieses obige Programm demonstriert eine Anweisung, in der der Programmierer das Werfen eines Fehlerobjekts (mit **throw**) in die umgebende Methode veranlasst.
Im Gegensatz zur Division durch 0 (wo die JVM das Fehlerobjekt wirf), veranlasst hier also der Programmierer durch die Verwendung des Bezeichnes **throw** das Werfen eines Fehlerobjekts!
In der umgebenden Methode (hier also main) muss dann das Fehlerobjekt mit try .. catch eingefangen werden oder wiederum an die dort umgebende Methode (mit throws) weitergeworfen werden.

1.1.9.2 Fall: Programmierer benutzt try ... catch

1.1.9.2.1 1. Fall

Wenn die Anweisungen des try-Bereichs ausgeführt sind, ohne dass ein Fehler passiert (kein Fehlerobjekt erzeugt wird), dann macht das Programm nach dem letzten catch-Block weiter (bzw. nach dem finally-Teil, falls es ihn gibt. Der finally-Teil wird vorher aber noch abgearbeitet).

1.1.9.2.2 2. Fall

Wenn bei einer Anweisungen des try-Bereichs ein Fehler geschieht, (ein Fehlerobjekt erzeugt wird), dann sucht das Programm (**ohne** die weiteren Anweisungen des try-Bereichs abzuarbeiten) sofort den **ersten** Ausnahme-Handler, dessen Parameter zu dem geworfenen Fehlerobjekt **passt** (es werden also nicht mehrere Ausnahme-Handler abgearbeitet). Das bedeutet, dass der Klassentyp des Fehlerobjekts und der Klassentyp des Parameters (in catch) gleich sind, bzw. der Klassentyp des Fehlerobjekts eine Unterklasse des Klassentyps des Parameters ist (siehe Konvertierung beim Methodenaufruf: das Objekt wird vor der Übergabe an den Parameter automatisch in das Objekt der Oberklasse konvertiert).
Dann macht das Programm nach dem letzten catch- Bezeichner weiter (bzw. nach dem finally-Teil, falls es ihn gibt. Der finally-Teil wird vorher aber noch abgearbeitet)).
Es wird also nicht notwendigerweise das Programm beendet!!

Beispiel:

```
public class MainException6{
  public static void main(String[] args){
    double erg;
    try{
      erg = 15 / 0;
    }
    catch(Exception mye){      // Zeile x1
      System.out.println("Division durch 0 unmöglich");
    }
    catch(Throwable myt){      // Zeile x2
      System.out.println("Irgendein anderer Fehler");
    }
  }
}
```

Bemerkungen:
1) Die Anweisung `z=15/0` wirft ein Fehlerobjekt vom Klassentyp ArithmeticException aus. Da aber ArithmeticException eine Unterklasse von RuntimeException und RuntimeException eine Unterklasse von Exception ist, wird dieses Fehlerobjekt dem ersten passenden Parameter, nämlich mye, zugewiesen.

2) Hätte man im obigen Beispiel Zeile x1 mit Zeile x2 vertauscht,

```
catch(Throwable myt){      // Zeile x2
...
catch(Exception mye){      // Zeile x1
```

würde der Compiler eine Fehlermeldung bringen, da das Programm dann nie zu Zeile x1 kommen würde: Da jedes geworfene Fehlerobjekt vom Klassentyp Throwable ist, würde es sofort vom ersten passenden Ausnahme-Handler "eingefangen" werden (das ist hier Zeile x2).

1.1.9.2.3 3. Fall

Wenn die Anweisungen des try-Bereichs ausgeführt sind und ein Fehler passiert (ein Fehlerobjekt erzeugt wird) und das geworfenen Fehlerobjekt zu keinem Parameter eines Ausnahme-Handlers passt, dann wird das Fehlerobjekt an die umgebende aufrufende Methode weitergeworfen (falls ein finally-Teil existiert, wird dieser noch ausgeführt).

1.1.10 Eigene Fehlerklassen basteln

Bei der ganzzahligen Division durch Null (z.B. 16/0) wird automatisch, d.h. ohne Zutun des Programmierers, ein Fehlerobjekt erzeugt. Der Programmierer kann aber auch durch "throw" selbst ein Fehlerobjekt einer schon existierenden Fehlerklasse, wie z.B. ArithmeticException oder NullPointerException, werfen.
Außerdem kann der Programmierer aber auch selbst eigene Fehlerklassen basteln und Objekte davon werfen.

In dem Beispiel unten soll in einer Methode festgestellt werden, ob in einer Zeichenkette ein ? enthalten ist. Wenn dies der Fall ist, kann diese Zeichenkette nicht als ein Verzeichnisname benutzt werden, da der Explorer in Windows dies nicht zulässt.
In diesem Fall soll dann ein Objekt einer selbst gebastelten Klasse geworfen werden.
Diese Fehlerklasse enthält im Wesentlichen eine Methode, die eine entsprechende Fehlermeldung auf dem Bildschirm ausgibt.

Bemerkungen:
1) "throw" veranlasst die Unterbrechung der "normalen" Abarbeitungsfolge des Programms: Es wird dann sofort (d.h. Anweisungen, die throw innerhalb derMethode folgen würden, wie z.B. return, werden nicht mehr ausgeführt) der entsprechende Ausnahme-Handler gesucht, der zu dem geworfenen Fehlerobjekt passt.

2) Die selbstgebastelte Klasse muss Unterklasse der Klasse Throwable oder einer Unterklasse davon sein.

1.1.11 Aufgaben

1) Programmieren Sie einen Kindertaschenrechner mit folgenden Eigenschaften:
a) Der Kindertaschenrechner soll bei Eingabe einer negativen Zahl sofort eine entsprechende Meldung bringen und dann das Programm sofort beenden (andere Idee: den Anwender die Eingabe wiederholen lassen).

b) Der Kindertaschenrechner soll im Fall eines negativen bzw. nicht ganzzahligen Ergebnisses sofort eine entsprechende Meldung bringen und dann das Programm sofort beenden (andere Idee: den Anwender die Eingabe wiederholen lassen).

c) Der Taschenrechner soll bei Division durch Null sofort eine entsprechende Meldung bringen und dann das Programm sofort (ohne Rechnung) beenden.

Bemerkungen:
1) siehe nächste Seite:
Die Methode indexOf (int zeichen) der Klasse String sucht das erste Vorkommen eines Zeichens (einer Zeichenkette) und gibt den zugehörigen Index zurück. Falls dieses Zeichens (Zeichenkette) nicht vorkommt, liefert die Methode -1 zurück.

1.1.11.1 Beispiel

```
public class MainException7 {
  public static void main(String[] args){
    Ordner v = new Ordner("Home?");
    try{
      v.checkOrdnerName();
    }
    catch(MyException_FalscherOrdnerName t){
      t.printFehlermeldung();
    }
  }
}

class Ordner{
  private String ordnerName;

  public Ordner(String pname){
    setOrdnerName(pname);
  }

  public void setOrdnerName(String pname){
    ordnerName = pname;
  }

  public String getOrdnerName(){
    return(ordnerName);
  }

  public void printAll(){
    System.out.println("Name des Ordners="+ordnerName);
  }

  public boolean checkOrdnerName() throws
                                MyException_FalscherOrdnerName{
    int zahl = (int)('?');
    if(ordnerName.indexOf(zahl)!=-1){
      throw new MyException_FalscherOrdnerName("Ordername
                                    "+ordnerName+" enthält ?");
    }
    return(true);
  }
}

class MyException_FalscherOrdnerName extends Throwable{
  private String zk;

  public MyException_FalscherOrdnerName(String pzk){
    zk=pzk;
  }

  public void printFehlermeldung(){
    System.out.println("Fehlermeldung: "+zk);
  }
}
```

Aufgaben

I) Mit den Methoden nextInt() bzw. nextDouble() der Klasse Scanner können Zahlen über Tastatur eingegeben werden.
Können dabei Fehlerobjekte geworfen werden, bzw. wie müssen diese verarbeitet werden (vom Programmierer eingefangen oder weitergeworfen werden)?
Siehe Java-Doku.

II) In der Java-Doku ist einer der RandomAccessFile-Konstrukturen wie folgt deklariert:

```
public RandomAccessFile(String name,
                        String mode)
                 throws FileNotFoundException
```

Weiter unten in der Doku steht:
Throws:
- IllegalArgumentException
- FileNotFoundException
- SecurityException

Frage:
Warum kann dieser Konstruktor 3 verschiedene Exceptions werfen?
Laut der Deklaration oben wird aber nur eine Exception mit throws geworfen.
Und die Exceptions IllegalArgumentException und SecurityException sind nicht Unterklassen von FileNotFoundException.

Auszug aus den Java-Doku: (<--- bedeutet Oberklasse von)

---

java.lang.Object <--- java.lang.Throwable <--- java.lang.Exception <---
java.lang.RuntimeException <--- java.lang.IllegalArgumentException

---

java.lang.Object <--- java.lang.Throwable <--- java.lang.Exception <---
java.io.IOException <--- java.io.FileNotFoundException

---

java.lang.Object <--- java.lang.Throwable <--- java.lang.Exception <---
java.lang.RuntimeException <--- java.lang.SecurityException

III) Ist der fianally-Bezeichner überflüssig, oder ist es nur programmtechnisch eleganter mit ihm zu arbeiten? Vergleichen Sie die zwei folgenden Programme:

```
//Methode 1
public void f1()throws
                  Throwable{
  try{
    öffne_Datei
    bewege_Dateizeiger
  }
  finally{
    schliesse_Datei
  }
}
```

```
//Methode 2
public void f2() throws
                  Throwable{
  try{
    öffne_Datei
    bewege_Dateizeiger
    schliesse_Datei
  }
  catch (Exception ex){
    schliesse_Datei
    throw ex;
  }
}
```

Bem:
try...catch ohne catch mit finally ist möglich (siehe früher).

Lösungen:
II) (siehe oben die Bedingungen über einzufangende und nicht einzufangende Fehler).

1) Wenn in der Methode
public RandomAccessFile(String name, String mode) throws FileNotFoundException
ein Ausnahmeobjekt der Klasse **IllegalArgumentException** oder **SecurityException** geworfen wird, also Klassen der Oberklasse RuntimeException, dann müssen diese nicht eingefangen oder mit throws weitergeworfen werden.

2) Wenn dagegen in der Methode
public RandomAccessFile(String name, String mode) throws FileNotFoundException
ein Ausnahmeobjekt der Klasse **FileNotFoundException** geworfen wird, muss dies entweder mit try .. catch eingefangen oder mit throws weitergeworfen werden.
Wie man in der Deklaration von
public RandomAccessFile(String name, String mode) throws **FileNotFoundException**
sieht, wird es nicht eingefangen, sondern mit throws weitergeworfen.

III)

```
//Methode 1
public void f1()throws
                Throwable{
  try{
    öffne_Datei
    bewege_Dateizeiger
  }
  finally{
    schliesse_Datei
  }
}
```

```
//Methode 2
public void f2() throws
                  Throwable{
  try{
    öffne_Datei
    bewege_Dateizeiger
    schliesse_Datei
  }
  catch (Exception ex){
    schliesse_Datei
    throw ex;
  }
}
```

1) Diskussion
finally ist zwar überflüssig, aber es hat Vorteile:

Fall1:
Dateizeiger wird über das Dateiende hinaus verschoben:
Deshalb wird eine Exception ausgelöst.
In der Methode1 wird vor dem Werfen des Fehlerobjekts noch die Anweisung im finally-Teil ausgeführt (also schliesse Datei). Dann wird das Fehlerobjekt in die umgebende Methode geworfen (weil es kein catch gibt, muß im Methodenkopf der umgebenden Methode "throws" vorkommen).
In der Methode2 wird das Fehlerobjekt geworfen und im catch-Teil eingefangen, d.h. dort wird dann weitergemacht mit schliesse_Datei und throw ex. Dieses mit throw ex erzeugte Fehlerobjekt wird dann in die umgebende Methode geworfen.
D.h. Methode1 und Methode2 machen semantisch das Gleiche.

Fall2:
Dateizeiger wird nicht über das Dateiende hinaus verschoben. Methode1 und Methode2 machen dann die folgenden Anweisungen:
öffne_Datei, bewege_Dateizeiger, schliesse_Datei
D.h. Methode1 und Methode2 machen semantisch das Gleiche.

2)
Methode2 verstösst aber gegen DRY (don't repeat yourself), weil doppelter Code produziert wird (nämlich schliesse_Datei).