Kapitel 12

Fehler und Ausnahmen

Ziele

- Fehlerquellen in Programmen und bei der Programmausführung kennenlernen
- Das Java-Konzept der Ausnahmen als Objekte verstehen
- Ausnahmen auslösen können
- Ausnahmen behandeln können

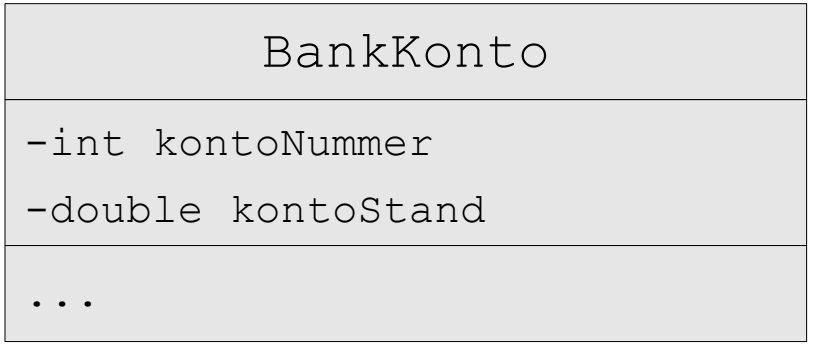
Fehlerhafte Programme

Ein Programm kann aus vielen Gründen unerwünschtes Verhalten zeigen.

- Logische Fehler beim Entwurf, d.h. bei der Modellierung des Problems: Entwurf entspricht nicht den Anforderungen.
- Fehler bei der Umsetzung des Entwurfs in ein Programm
  - Programm bricht ab wegen Division durch 0
  - Algorithmen falsch implementiert
  - Programm entspricht nicht dem Entwurf
- Ungenügender Umgang mit außergewöhnlichen Situationen
  - Fehlerhafte Benutzereingaben, z.B. Datum 31.11.2010
  - Abbruch der Netzwerkverbindung
  - Dateien können nicht gefunden werden, …

Entwurfsfehler

- **Anforderung:** … Eine Bank kann mehrere Bankkonten verwalten . …
- Fehlerhafter Entwurf:



Probleme:

- Anforderungen sind nur selten eindeutig und präzise.
- Änderungen an Anforderungen sind die Regel.
- Entwurf wird im Rahmen eines Sofware-Entwicklungsprozesses erstellt; auf Programmiersprachenebene gibt es nur begrenzte Möglichkeiten, Entwurfsfehler zu erkennen und ihnen entgegenzuwirken.

Programmierfehler (1)

Viele Fehler findet schon der Compiler zur Übersetzungszeit:

- **Syntaxfehler**: Das Programm entspricht nicht der Java-Grammatik.

  Beispiel: `while x > 0 {...}` statt `while (x > 0) {...}`

- **Typfehler**: Ein Ausdruck oder eine Anweisung hat einen falschen Typ.

  Beispiel: `while (x > true) {...}`

Andere Fehler erkennt man erst daran, dass die Ausführung des Programms unerwartet abbricht (**Laufzeitfehler**):

- **Laufzeitfehler wegen falscher Implementierung:**

  Division durch 0, Zugriff auf nichtexistierenden Index in einem Array, Methodenaufruf für die leere Referenz `null`, …

- **Laufzeitfehler wegen ungenügender Behandlung von Ausnahmesituationen:**

  ungültige Benutzereingaben, Netzwerkverbindung unterbrochen, …

Programmierfehler (2)

Häufig gibt es auch **logische Fehler**:

- Das Programm implementiert nicht den Entwurf.
- Das Programm liefert falsche Ergebnisse.
- Das Programm liefert gar keine Ergebnisse.

Es gibt eine Vielzahl von Methoden, die sich mit der **logischen Korrektheit** von Programmen beschäftigen, z.B.:

- Konstruktive Qualitätssicherungsmaßnahmen
- Entwicklungsmethoden (z.B. Design by Contract)
- Systematisches Testen
- Programminspektionen
- Automatisierte Fehlersuche
- Korrektheitsbeweise (z.B. Hoare-Logik)

Robuste Programme

- Wir beschäftigen uns hier nicht damit, wie man logische Fehler in einem Programm finden kann.
- Wir konzentrieren uns hier im Wesentlichen auf die Vermeidung von Laufzeitfehlern und deren Behandlung.
- Unser Ziel ist es robuste Programme zu schreiben.

Definition:

Ein Programm heißt ***robust***, falls es für jede (auch fehlerhafte) Eingabe eine sinnvolle Reaktion produziert.

Ein robustes Programm muss also alle Ausnahmesituationen bei der Programmausführung **erkennen**, möglichst **vermeiden** aber zumindest sinnvoll **behandeln**.

Robuste Programme: Beispiel

- Folgendes Programm ist nicht robust.

```
int n = Input.readInt();  // Annahme: Input ist eine Klasse für Benutzereingaben
// Ausgabe der Zahlen von 100 bis 200 in n-er Schritten
for (int i=100; i < 200; i = i+n) {
    System.out.println(i);
}
```

  Es terminiert nicht bei Eingabe 0 und liefert ungewollte Ergebnisse bei negativen Eingaben.

- Möglichkeit das Programm robust zu machen:

```
int n = 0;
while (n <= 0) {
    n = Input.readInt();
}
// Ausgabe der Zahlen von 100 bis 200 in n-er Schritten
for (int i=100; i < 200; i = i+n) {
    System.out.println(i);
}
```

Ausnahmesituationen erkennen (1)

Beispiel: Löschen des ersten Elements in einer verketteten Liste.

```
class MyList {
   private ListElement first;
   …
   public double removeFirst_0() {
        double value = first.getValue();
        first = first.getNext();
        return value;
    }
}
```

```
class Test {
   public static void main(String args[])  {
      MyList list = …;
      double d = list.removeFirst_0();
      System.out.println(" Entferntes
Element hatte den Wert " + d);
      …
   }
}
```

- Mögliche Ausnahmesituationen werden in der Methode `removeFirst_0` **nicht** erkannt.
- Es kann zu einer `NullPointerException` kommen falls `(first == null)` ist.
- Die Methode und das Programm werden dann abgebrochen und die `NullPointerException` wird auf der Konsole gemeldet.

Ausnahmesituationen erkennen (2)

```
class MyList {
  ...
  public double removeFirst_1() {
        if (first == null) {
            return 0;
        }
        double value = first.getValue();
        first = first.getNext();
        return value;
  }
}
```

```
class Test {
   public static void main(String args[])  {
      MyList list = …;
      double d = list.removeFirst_1();
      System.out.println(" Entferntes
Element hatte den Wert " + d);
      …
   }
}
```

Eine Ausnahmesituation wird in der Methode `removeFirst_1` zwar erkannt, aber nicht adäquat dem Aufrufer gemeldet.

Ausnahmen auslösen

```
class MyList {
  …
  public double removeFirst_2() {
      if (first == null) {
        throw new NoSuchElementException(
"Kann in leerer Liste nicht löschen.");
      }
      double value = first.getValue();
      first = first.getNext();
      return value;
  } }
```

```
class Test {
  public static void main(String args[])  {
    MyList list = …;
    double d = list.removeFirst_2();
    System.out.println(" Entferntes
Element hatte den Wert " + d);
    …
  }
}
```

- Eine Ausnahmesituation wird in der Methode `removeFirst_2` erkannt und es wird eine Ausnahme ausgelöst („geworfen“).
- Die Ausnahme enthält eine individuelle Information, die die Fehlersituation beschreibt.
- Wird eine Ausnahme ausgelöst, dann werden die Methode und das Programm abgebrochen und die Fehlermeldung wird auf der Konsole gemeldet.
- Alternativ kann die Ausnahme auch **behandelt** werden (vgl. später).

Vermeiden von Ausnahmen

```
class MyList {
  …
  public double removeFirst_2() {
      if (first == null) {
        throw new NoSuchElementException(
"Kann in leerer Liste nicht löschen.");
      }
      double value = first.getValue();
      first = first.getNext();
      return value;
  } }
```

```
class Test {
  public static void main(String args[])  {
    MyList list = …;
    if (list.size() > 0) {
      double d = list.removeFirst_2();
      System.out.println(" Entferntes
Element hatte den Wert " + d);
    }
    …
  }
}
```

- Die Ausnahmesituation wird hier vermieden, indem die `main`-Methode vor dem Aufruf der Methode `removeFirst_2` testet, ob die Liste nicht leer ist.
- Es kann (diesbezüglich) keine Ausnahme mehr ausgelöst werden.
- Die Methode `removeFirst_2` beinhaltet jedoch weiterhin eine Ausnahmeerkennung und ggf. eine Ausnahmeauslösung, da sie nicht sicher ist, ob sie korrekt aufgerufen wird („defensive“ Programmierung).

Fehler- und Ausnahmenklassen in Java

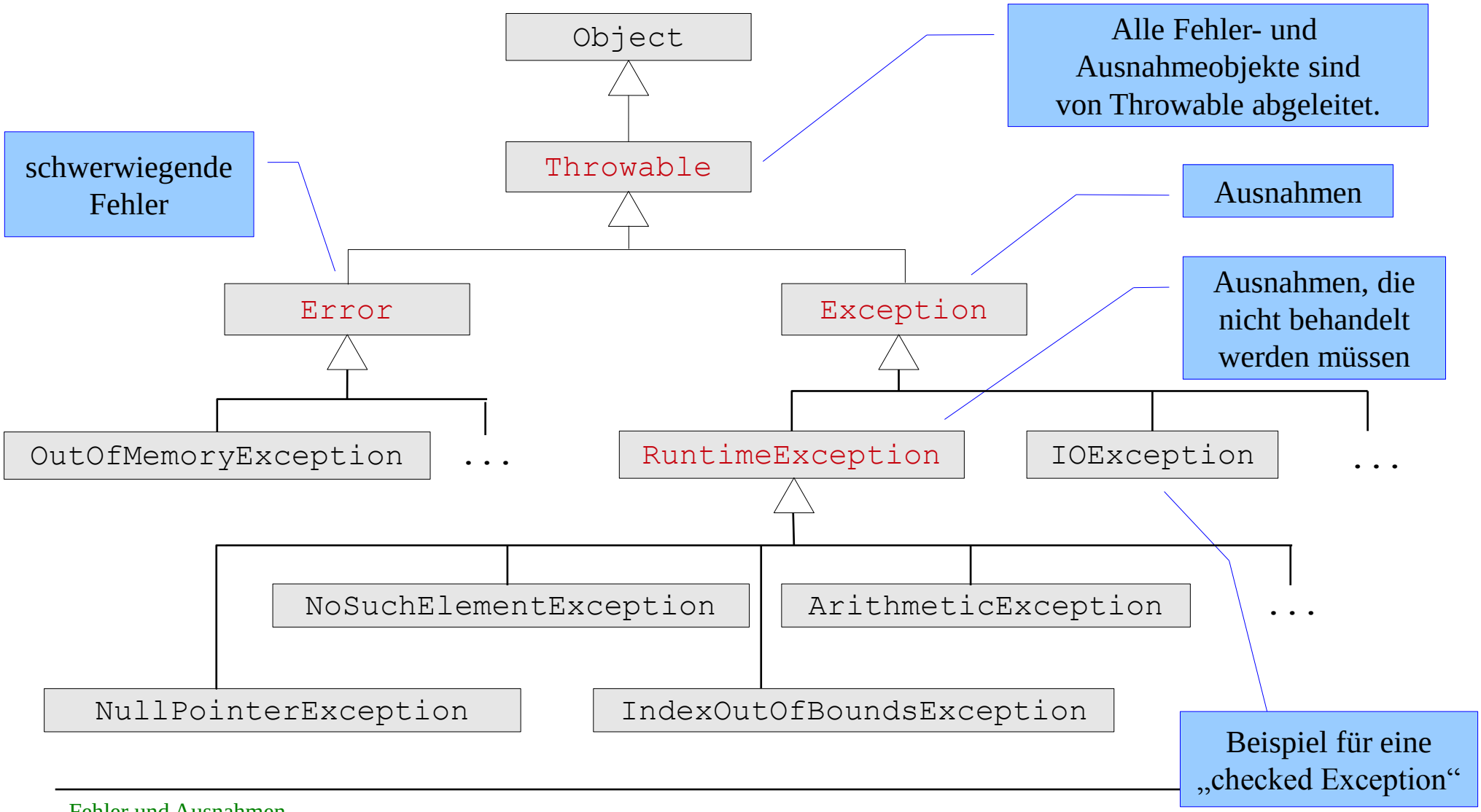
In Java unterscheidet man zwischen Fehlern und Ausnahmen, die beide durch **Objekte** repräsentiert werden.

- Fehler sind Instanzen der Klasse `Error`
- Ausnahmen sind Instanzen der Klasse `Exception`
  - Ausnahmen, die nicht unbedingt vom Programmierer behandelt werden müssen: „**unchecked Exceptions**“ (Instanzen der Klasse `RuntimeException`)
  - Ausnahmen, die vom Programmierer behandelt werden müssen: „**checked Exceptions**“ (alle anderen Instanzen von `Exception`)

Fehler deuten auf schwerwiegende Probleme hin und sollten nie behandelt werden.

Ausnahmen können vom Programmierer im Programm durch Ausnahmebehandlung abgefangen werden.

Vererbungshierarchie der Fehlerklassen



Die Klasse `Error` und ihre direkten Subklassen

java.lang

Class Error

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Error
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationFormatError, AssertionError, AWTError, CoderMalfunctionError, FactoryConfigurationError, FactoryConfigurationError, IOError, LinkageError, ServiceConfigurationError, ThreadDeath, TransformerFactoryConfigurationError, VirtualMachineError

---

```
public class Error
extends Throwable
```

An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The `ThreadDeath` error, though a "normal" condition, is also a subclass of `Error` because most applications should not try to catch it.

A method is not required to declare in its `throws` clause any subclasses of `Error` that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur.

Die Klasse `Exception` und ihre direkten Subklassen

java.lang

Class Exception

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, ClassNotFoundException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalAccessException, IllegalClassFormatException, InstantiationException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, InvocationTargetException, IOException, JAXBException, JMException, KeySelectorException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NoSuchFieldException, NoSuchMethodException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

Die Klasse `RuntimeException` und ihre direkten Subklassen

java.lang

Class RuntimeException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.lang.RuntimeException

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DataBindingException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, ImagingOpException, IncompleteAnnotationException, IndexOutOfBoundsException, JMRuntimeException, LSException, MalformedParameterizedTypeException, MirroredTypeException, MirroredTypesException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ProfileDataException, ProviderException, RasterFormatException, RejectedExecutionException, SecurityException, SystemException, TypeConstraintException, TypeNotPresentException, UndeclaredThrowableException, UnknownAnnotationValueException, UnknownElementException, UnknownTypeException, UnmodifiableSetException, UnsupportedOperationException, WebServiceException

Die Klasse `IOException` und ihre direkten Subklassen

java.io

Class IOException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.io.IOException

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

ChangedCharSetException, CharacterCodingException, CharConversionException, ClosedChannelException, EOFException, FileLockInterruptionException, FileNotFoundException, FilerException, HttpRetryException, IIOException, InterruptedIOException, InvalidPropertiesFormatException, JMXProviderException, JMXServerErrorException, MalformedURLException, ObjectStreamException, ProtocolException, RemoteException, SaslException, SocketException, SSLException, SyncFailedException, UnknownHostException, UnknownServiceException, UnsupportedDataTypeException, UnsupportedEncodingException, UTFDataFormatException, ZipException

Die Klasse `Throwable`

- Ausnahme- und Fehler-Objekte enthalten Informationen über Ursprung und Ursache des Fehlers.
- Die Klasse `Throwable`, von der alle Fehlerklassen abgeleitet sind, verwaltet solche Informationen, z.B.:
  - eine **Nachricht** zur Beschreibung des aufgetretenen Fehlers
  - einen **Schnappschuss** des Aufrufstacks zum Zeitpunkt der Erzeugung des Objekts
- Nützliche Methoden in Throwable:
  - `String getMessage()`: gibt die Fehlermeldung zurück
  - `void printStackTrace()`: gibt den Aufrufstack des Fehlers aus

Auslösung einer RuntimeException und Ausgabe des Aufrufstacks

```
public class Div0 {
  /** Die Methode m loest wegen der Division durch 0
   * eine ArithmeticException aus:   */
  public static void m() {
    int d = 0;
    int a = 42 / d;
    System.out.println("d= " + d);
    System.out.println("a= " + a);
   }
  public static void main(String args[]) {
    m();
  }
}
```

Java-Ausgabe mit Aufrufstack:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Div0.m(Div0.java:6)
at Div0.main(Div0.java:11)
```

Der Aufrufstack enthält die Folge der Methodenaufrufe, die zum Fehler geführt haben.

Kontrolliertes Auslösen von Ausnahmen

- Mittels der **`throw`**-Anweisung kann man eine Ausnahme auslösen.
- **Syntax**:

```
throw exp;
```

- Der Ausdruck `exp` muss eine Instanz einer von `Throwable` abgeleiteten Klasse (d.h. eine Ausnahme oder ein Fehlerobjekt) bezeichnen.
- Die Ausführung einer **`throw`**-Anweisung stoppt den Kontrollfluss des Programms und löst die von `exp` definierte Ausnahme aus.
- Es kommt zu einem Abbruch des Programms, wenn die Ausnahme nicht in einer übergeordneten Methode abgefangen und behandelt wird.

Kontrolliertes Auslösen von Ausnahmen: Beispiel

- Löschen des ersten Elements einer verketteten Liste

```
public double removeFirst_2() {
  if (first == null) {
    throw new NoSuchElementException(
            "Kann in leerer Liste nicht löschen.");
    }
  double value = first.getValue();
  first = first.getNext();
  return value;
}
```

Wird nicht ausgeführt wenn `first == null`.

- Die Anweisung **new** NoSuchElementException("Kann nicht …"); erzeugt ein neues Ausnahmeobjekt mit gegebener Fehlernachricht.
- Dieses Objekt hat zunächst den gleichen Status wie jedes andere Objekt auch.
- Erst mit throw wird die Ausnahme tatsächlich ausgelöst. Die nachfolgenden Anweisungen werden dann nicht mehr ausgeführt (wie bei `return`).

Geprüfte Ausnahmen (Checked Exceptions)

- In Java gibt es **geprüfte** (**checked**) und **ungeprüfte** (**unchecked**) Ausnahmen.
- Gibt es in einem Methodenrumpf eine `throw`-Anweisung mit einer geprüften Ausnahme, dann muss das im Methodenkopf mit … **`throws`** … explizit als möglich deklariert werden.
- Geprüfte Ausnahmen müssen vom Aufrufer der Methode entweder behandelt werden oder wieder im Methodenkopf deklariert werden.
- Spätestens in der `main`-Methode muss eine geprüfte Ausnahme behandelt werden.
- Beispiele:

```
import java.io.IOException;
  …
  public void m() throws IOException {
    if (…) {
     throw new IOException();
    }
  }
  public void n() throws IOException {
    m();
  }
```

Man muss deklarieren, dass in dieser Methode die Ausnahme `IOException` auftreten kann.

Da die `IOException`, die beim Aufruf von `m()` auftreten kann, hier nicht behandelt wird, muss die Methode selbst wieder eine `throws`-Klausel deklarieren

Ungeprüfte Ausnahmen (Unchecked Exceptions)

- Ungeprüfte Ausnahmen sind genau die Instanzen von `RuntimeException`.
- Ungeprüfte Ausnahmen müssen weder behandelt noch im Methodenkopf explizit als möglich deklariert werden.
- Beispiele: `ArithmeticException, NullPointerException, …`

Motivation für die Unterscheidung geprüft/ungeprüft:

- Geprüfte Ausnahmen werden verwendet für Ausnahmefälle, die sich vom Programmierer nicht verhindern lassen. Diese Fälle sollten im Programm immer behandelt werden. Der Compiler prüft, dass kein Fall vergessen wird.
- Ungeprüfte Ausnahmen können dagegen vom Programmierer, zumindest prinzipiell, vollständig verhindert werden.

Benutzerdefinierte Ausnahmeklassen

- Die Missachtung logischer Anforderungen kann zu anwendungsspezifischen Ausnahmen führen.
- Mittels Vererbung kann man eigene Ausnahmeklassen definieren.

Beispiel:

- Klassen `BankKonto` und `SparKonto`. Es soll nicht möglich sein, ein `SparKonto` zu überziehen.
- Wir definieren dazu eine (checked) Exception, die beim Versuch das `SparKonto` zu überziehen, geworfen werden soll:

```
public class KontoUngedecktException extends Exception {
  private double abhebung;
  public KontoUngedecktException(String msg, double abhebung) {
    super(msg); // Konstruktor von Exception nimmt Nachricht
    this.abhebung = abhebung;
  }
  public double getAbhebung() {
    return abhebung;
  }}
```

Auslösen einer benutzerdefinierten Ausnahme

```
public class BankKonto {
   …
   public void abheben(double x) throws KontoUngedecktException {
      kontoStand = kontoStand - x;
   }
}

public class SparKonto extends BankKonto {
   …
   public void abheben(double x) throws KontoUngedecktException {
      if (getKontoStand() < x) {
         throw new KontoUngedecktException("Sparkonten dürfen nicht überzogen werden.", x);
      }
      super.abheben(x);
   }
}
```

Behandlung von Ausnahmen

Ausnahmebehandlung geschieht in Java mit Hilfe der **try**-Anweisung. Damit können Ausnahmen **abgefangen** werden.

```
try {
   // Block fuer „normalen“ Code
} catch (Exception1 e) {
   // Ausnahmebehandlung fuer Ausnahmen vom Typ Exception1
} catch (Exception2 e) {
   // Ausnahmebehandlung fuer Ausnahmen vom Typ Exception2
}
```

- Zunächst wird der **try**-Block normal ausgeführt.
- Tritt im **try**-Block *keine* Ausnahmesituation auf, so werden die beiden Blöcke zur Ausnahmebehandlung ignoriert.
- Tritt im **try**-Block eine Ausnahmesituation auf (z.B. wegen **throw**), so wird die Berechnung dieses Blocks abgebrochen.
  - Ist die Ausnahme vom Typ `Exception1` oder `Exception2`, so wird der Block nach dem jeweiligen **catch** ausgeführt.
  - Ansonsten ist die Ausnahme unbehandelt; das Programm verhält sich wie bei der Ausführung des **try**-Blocks allein.

Ausnahmebehandlung bei fehlerhafter GUI-Eingabe (1)

```
public class ExceptionTestFrame extends JFrame implements
ActionListener {

  private JButton testButton;
  private JTextArea ausgabeBereich;

  public ExceptionTestFrame() {
    ...
    this.testButton.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    if (source == this.testButton) {
      this.test();
    }
  }
```

Ausnahmebehandlung bei fehlerhafter GUI-Eingabe (2)

```
private void test() {
  int wert = 0;
  boolean inputOk = false;
  while (!inputOk) {
    String einlesenWert = JOptionPane.showInputDialog("Wert?");
    try {
      wert = Integer.parseInt(einlesenWert);
      inputOk = true;
    } catch (NumberFormatException e) {
      this.ausgabeBereich.setText("Falsche Eingabe: Kein Integer!");
    }
  }//Ende while
  this.ausgabeBereich.setText("Eingabe war" + wert);
}//Ende Methode test
}//Ende Klasse ExceptionTestFrame
```

Behandlung von Ausnahmen: Beispiel Division durch 0

```
public class Div0 {
  /** Die Methode main loest wegen der Division durch 0
   * eine ArithmeticException aus:   */
  public static void main(String args[]) {
    try {
        int d = 0;
        int a = 42 / d;
    } catch (ArithmeticException e) {
        System.out.println("Division durch 0.");
        System.out.println("e.getMessage() liefert: " + e.getMessage());
    }
    System.out.println("Programm bricht nicht ab.");
  }
}
```

Ausgabe dieses Programms:

```
Division durch 0.
e.getMessage() liefert: / by zero
Programm bricht nicht ab.
```

Besser wäre zu vermeiden, dass durch 0 geteilt wird!

Behandlung von Ausnahmen: Beispiel Konto

Geprüfte Ausnahmen **müssen** abgefangen oder deklariert werden, z.B.:

```
public static void main(String[] args) {

   SparKonto konto = new SparKonto(5, 1); // 5 Euro, 1% Zinsen

   System.out.println("Wie viel wollen Sie abheben? ");
   double betrag = Input.readInt();

   try {
       konto.abheben(betrag);
   } catch (KontoUngedecktException e) {
       System.out.println(e.getMessage());
       System.out.println("Der Abhebungsbetrag" + e.getAbhebung() + "war zu hoch. ");
   }
}
```

Hinweis: Eine Division durch 0 im `try`-Block würde durch das `catch` nicht abgefangen und daher zum Programmabbruch führen.

Behandlung von Ausnahmen: finally

Manchmal möchte man nach der Ausführungs eines **try**-Blocks bestimmte Anweisungen ausführen, egal ob eine Ausnahme aufgetreten ist.

- Beispiel: Schließen einer im **try**-Block geöffneten Datei.
- Das kann man mit einem optionalen **finally**-Block erreichen, der in jedem Fall nach dem **try**-Block und der Ausnahmebehandlung ausgeführt wird.

```
try {
   // Block fuer „normalen“ Code
} catch (Exception1 e) {
   // Ausnahmebehandlung fuer Ausnahmen vom Typ Exception1
} catch (Exception2 e) {
   // Ausnahmebehandlung fuer Ausnahmen vom Typ Exception2
} finally {
   // Code, der in jedem Fall nach normalem Ende und nach
   // Ausnahmebehandlung ausgefuehrt werden soll.
}
```

Beispiel für `finally` (1)

Ablauf in einem Geldautomaten:

```
System.out.println("Wie viel wollen Sie abheben? ");
double betrag = Input.readInt();

try {
    konto.abheben(betrag);
} catch (KontoUngedecktException e) {
    System.out.println(e.getMessage());
    System.out.println("Der Abhebungsbetrag" + e.getAbhebung() + "war zu hoch. ");
} finally {
    System.out.println("Bitte entnehmen Sie ihre Karte.");
}
```

Beispiel für `finally` (2)

```
public class Div0 {
  /** Die Methode main loest wegen der Division durch 0
   * eine ArithmeticException aus:   */
  public static void main(String args[]) {
    try {
      int d = 0;
      int a = 42 / d;
    } catch (ArithmeticException e) {
      System.out.println("Division durch 0.");
    } finally {
      System.out.println("Wird in jedem Fall gemacht.");
    }
    System.out.println("Ende.");
  }
}
```

Ausgabe dieses Programms:

```
Division durch 0.
Wird in jedem Fall gemacht.
Ende.
```

Beispiel für `finally` (3)

```
public class Div0 {
   /** Die Methode main loest wegen der Division durch 0
    * eine ArithmeticException aus:   */
   public static void main(String args[]) {
      try {
         int d = 0;
         int a = 42 / d;
      } finally {
         System.out.println("Wird in jedem Fall gemacht.");
      }
      System.out.println("Ende.");
    }
}
```

Ausgabe dieses Programms:

```
Wird in jedem Fall gemacht.
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Div0.main(Div0.java:7)
```

Beispiel: Lesen von der Standardeingabe (Konsole)

Eine robuste Methode zum Einlesen einer Zeile von der Konsole:

```
public static String readString() {

  BufferedReader in = new BufferedReader(
                              new InputStreamReader(System.in));

  while (true) {

    try {

      return in.readLine();

    } catch (IOException e) {
      System.out.println("Fehler beim Einlesen: " + e.getMessage());
      System.out.println("Versuchen Sie es nochmal!");
    }

  }
}
```

Klasse mit Operationen zur Verarbeitung von Textströmen

Datenstrom von Bytes von der Konsole

Konvertiert `byte`-Strom in `char`-Strom

Liest nächste Zeile aus Eingabestrom

Bei einem IO-Fehler in `in.readLine()` kommt die **`return`**-Anweisung nicht zur Ausführung. Es wird stattdessen dieser Block zur Fehlerbehandlung ausgeführt und dann wird mit der `while`-Schleife weitergemacht (d.h. die Eingabe wiederholt).

Zusammenfassung

- Ausnahmen werden in Java durch Objekte dargestellt.
- Methoden können Ausnahmen auslösen implizit durch einen Laufzeitfehler oder explizit mit **`throw`** und damit „abrupt“ terminieren.
- Ausnahmen können mit **`catch`** behandelt werden, so dass sie nicht zu einem Abbruch des Programms führen.
- Wir unterscheiden geprüfte und ungeprüfte Ausnahmen.
- Geprüfte Ausnahmen müssen abgefangen werden oder im Kopf der Methode als möglich deklariert werden.
- In jedem Fall ist es am Besten Ausnahmen zu vermeiden.
- Defensive Programme sehen auch für vermeidbare Ausmahmesituationen das Werfen von Ausnahmen vor (was dann hoffentlich nie nötig ist).