

# Kapitel 12

---

## Fehler und Ausnahmen

## Ziele

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

## Fehlerquellen

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 entspricht nicht dem Entwurf (logischer Programmierfehler),
  - Algorithmen falsch implementiert (logischer Programmierfehler),
  - Programm bricht ab wegen vermeidbarem Laufzeitfehler; z.B. Division durch 0, falscher Arrayzugriff, ...
- **Ungenügender Umgang mit außergewöhnlichen Situationen, die nicht vermieden werden können.**
  - Fehlerhafte Benutzereingaben, z.B. Datum 31.11.2010
  - Fehlerhafter Dateizugriff, Abbruch der Netzwerkverbindung, ...

## Robuste Programme

- Wir beschäftigen uns hier nicht damit, wie man logische Fehler in einem Programm finden kann.
- Wir konzentrieren uns hier auf das Erkennen, Vermeiden und Behandeln von Ausnahmesituationen, die zu Laufzeitfehlern führen können.
- Unser Ziel ist es robuste Programme zu schreiben.

### **Definition:**

Ein Programm heißt ***robust***, falls es auch beim Auftreten von Fehlern sinnvoll reagiert.

## Ausnahmesituationen erkennen (1)

**Beispiel:** Berechnung der Fakultät.

```
class MyClass {  
  
    public static int fact(int n) {  
        if (n == 0) return 1;  
        else return n * fact(n-1);  
    }  
}
```

```
class Test {  
    public static void main(String args[]) {  
        int m = ...;  
        int k = MyClass.fact(m);  
        System.out.println("Fakultät von " + m  
+ " ist " + k);  
    }  
}
```

- Es kann zu Nichtterminierung kommen.
- Diese Ausnahmesituation wird in der Methode `fact` **nicht** „erkannt“.

## Ausnahmesituationen erkennen (2)

**Beispiel:** Berechnung der Fakultät.

```
class MyClass {  
  
    public static int fact_1(int n) {  
        if (n < 0) return -1;  
        else if (n == 0) return 1;  
        else return n * fact(n-1);  
    }  
}
```

```
class Test {  
    public static void main(String args[]) {  
        int m = ...;  
        int k = MyClass.fact_1(m);  
        System.out.println("Fakultät von " + m  
+ " ist " + k);  
    }  
}
```

Ausnahmesituationen ( $n < 0$ ) werden in der Methode `fact_1` zwar erkannt, aber nicht adäquat an den Aufrufer gemeldet.

## Ausnahmesituation erkennen und Ausnahme auslösen

```
class MyClass {  
  
    public static int fact_2(int n) {  
        if (n < 0) throw new  
        IllegalArgumentException("Eingabe  
        darf nicht kleiner als 0 sein.");  
        if (n == 0) return 1;  
        else return n * fact(n-1);  
    }  
}
```

```
class Test {  
    public static void main(String args[]) {  
        int m = ...;  
        int k = MyClass.fact_2(m);  
        System.out.println("Fakultät von " + m  
        + " ist " + k);  
    }  
}
```

- Eine Ausnahmesituation wird in der Methode `fact_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 individuell im main-Programm **behandelt** werden (vgl. später).

## Vermeiden von Ausnahmen

```
class MyClass {  
  
    public static int fact_2(int n) {  
        if (n < 0) throw new  
IllegalArgumentException("Eingabe  
darf nicht kleiner als 0 sein.");  
        if (n == 0) return 1;  
        else return n * fact(n-1);  
    }  
}
```

```
class Test {  
    public static void main(String args[]) {  
        int m = ...;  
        if (m >= 0) {  
            int k = MyClass.fact_2(m);  
            System.out.println("Fakultät von " +  
m + " ist " + k);  
        }  
    }  
}
```

- Die Ausnahmesituation wird hier vermieden, indem die `main`-Methode vor dem Aufruf der Methode `fact_2` testet, ob das Argument größer gleich 0 ist.
- Es kann (diesbezüglich) keine Ausnahme mehr ausgelöst werden.
- Die Methode `fact_2` beinhaltet jedoch weiterhin eine Ausnahmeerkennung mit 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.

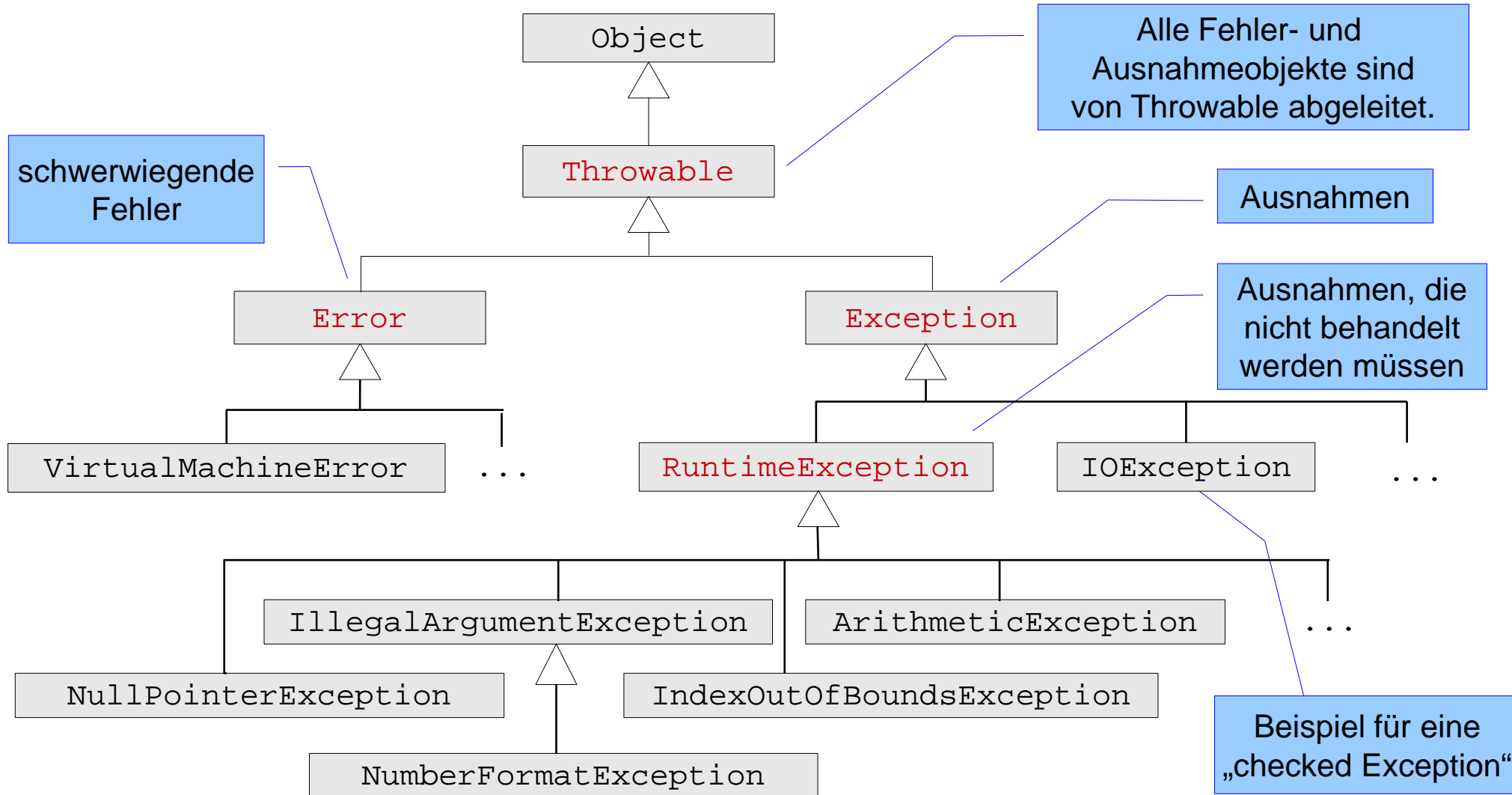
- Fehler sind Instanzen der Klasse `Error`
- Ausnahmen sind Instanzen der Klasse `Exception`

Fehler deuten auf schwerwiegende Probleme der Umgebung hin und können nicht sinnvoll behandelt werden, z.B. `VirtualMachineError`.

Das Auftreten solcher Fehler sollte, wo möglich, vermieden werden (wie z.B. in der Methode `fact_2`).

Ausgelöste Ausnahmen können vom Programmierer im Programm individuell behandelt 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:

[AccessDeniedException](#), [ActivationException](#), [AlreadyBoundException](#), [ApplicationException](#), [AWTException](#), [BackingStoreException](#), [BadAttributeValueExpException](#), [BadBinaryOpValueExpException](#), [BadLocationException](#), [BadStringOperationException](#), [BrokenBarrierException](#), [CertificateException](#), [ClassNotFoundException](#), [CloneNotSupportedException](#), [DateFormatException](#), [DatatypeConfigurationException](#), [DestroyFailedException](#), [ExecutionException](#), [ExpandVetoException](#), [FontFormatException](#), [GeneralSecurityException](#), [GSSEException](#), [IllegalAccessException](#), [IllegalClassFormatException](#), [InstantiationException](#), [InterruptedException](#), [IntrospectionException](#), [InvalidApplicationException](#), [InvalidMidiDataException](#), [InvalidPreferencesFormatException](#), [InvalidTargetObjectTypeException](#), [InvocationTargetException](#), [IOException](#), [JAXBException](#), [JMEException](#), [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](#), [UnsupportedAudioFormatException](#), [UnsupportedCallbackException](#), [UnsupportedFlavorException](#), [UnsupportedLookAndFeelException](#), [URISyntaxException](#), [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`](#), [`LSEException`](#), [`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 Throwable

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

## Auslösung einer ArithmeticException 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[]) {  
        this.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.

- **Beispiel:**

```
throw new IllegalArgumentException( "... " );
```

- Die Ausführung einer **throw**-Anweisung stoppt den Kontrollfluss der Methode und löst die von `exp` definierte Ausnahme aus. Die nachfolgenden Anweisungen im Rumpf der Methode werden nicht mehr ausgeführt (wie bei **return**).
- Es kommt zu einem Abbruch des Programms, wenn die Ausnahme nicht in einer übergeordneten Methode abgefangen und behandelt wird.



## Geprüfte Ausnahmen (Checked Exceptions)

- Geprüfte Ausnahmen sind in Java alle Instanzen der Klasse `Exception`, die **nicht** Objekte der Klasse `RuntimeException` sind.
- Kann während der Ausführung einer Methode eine geprüfte Ausnahme auftreten, dann muss diese entweder in der Methode behandelt werden (vgl. unten) oder es muss im Methodenkopf mit **throws Ausnahmetyp** explizit darauf hingewiesen werden (wird vom Compiler geprüft).
- Beispiel:

```
import java.io.IOException;
...
public void m() throws IOException {
    if (...) {
        throw new IOException();
    }
}
public void n() throws IOException {
    this.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`.
- Beispiele: `ArithmeticException`, `NullPointerException`, ...
- Ungeprüfte Ausnahmen müssen nicht im Methodenkopf explizit deklariert werden (sie können es aber).
- Ungeprüfte Ausnahmen müssen nicht behandelt werden (sie können es aber).

## Benutzerdefinierte Ausnahmeklassen

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 abhebungsbetrag;

    public KontoUngedecktException(String msg, double abhebungsbetrag) {
        super(msg); // Konstruktor von Exception nimmt Nachricht
        this.abhebungsbetrag = abhebungsbetrag;
    }

    public double getAbhebungsbetrag() {
        return this.abhebungsbetrag;
    }
}
```

## Auslösen einer benutzerdefinierten Ausnahme

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

```
public class SparKonto extends BankKonto {  
    ...  
    public void abheben(double x) throws KontoUngedecktException {  
        if (this.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, 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.

## Behandlung von Ausnahmen: Beispiel Konto

```
public static void main(String[] args) {  
  
    SparKonto konto = new SparKonto(5, 1); // 5 Euro, 1% Zinsen  
  
    String einleseBetrag = JOptionPane.showInputDialog("Betrag zum Abheben?");  
    double betrag = Double.parseDouble(einleseBetrag);  
  
    try {  
        konto.abheben(betrag);  
    } catch (KontoUngedecktException e) {  
        System.out.println(e.getMessage());  
        System.out.println("Der Abhebungsbetrag " + e.getAbhebungsbetrag() + " war zu hoch. ");  
    }  
}
```

## Behandlung von Ausnahmen: finally

Manchmal möchte man nach der Ausführung 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 **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 ausgeführt werden soll (nach normalem  
       Ende, nach Ausnahmebehandlung und nach Auslösung einer nicht  
       behandelten Ausnahme).  
    */  
}
```

## Beispiel für `finally`

Ablauf in einem Geldautomaten:

```
public static void main(String[] args) {  
  
    SparKonto konto = new SparKonto(5, 1); // 5 Euro, 1% Zinsen  
  
    String einleseBetrag = JOptionPane.showInputDialog("Betrag zum Abheben?");  
    double betrag = Double.parseDouble(einleseBetrag);  
  
    try {  
        konto.abheben(betrag);  
    } catch (KontoUngedecktException e) {  
        System.out.println(e.getMessage());  
        System.out.println("Der Abhebungsbetrag " + e.getAbhebungsbetrag() + " war zu hoch. ");  
    } finally {  
        System.out.println("Bitte entnehmen Sie ihre Karte.");  
    }  
}
```



## Ausnahmebehandlung bei fehlerhafter GUI-Eingabe

```
public void actionPerformed(ActionEvent e) {
    double betrag;
    Object source = e.getSource();
    if (source == this.einzahlenButton) {
        boolean inputOk = false;
        while (!inputOk) {
            try {
                betrag = Double.parseDouble(this.betragField.getText());
                inputOk = true;
            } catch (NumberFormatException e) {
                this.ausgabeBereich.setText("Falsche Eingabe: Kein Double!");
            } //end try-catch
        } //end while
        this.konto.einzahlen(betrag);
        this.kontoStandField.
            setText(Double.toString(this.konto.getKontoStand()));
    } //end if
    else if (source == this.abhebenButton) { . . .
    }
}
```

## 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 wiederum deklariert werden.
- In jedem Fall ist es am Besten Ausnahmen zu vermeiden, wenn es in der Macht des Programmierers liegt.
- Defensive Programme sehen auch für vermeidbare Ausnahmesituationen das Werfen von Ausnahmen vor (was dann hoffentlich nie nötig ist).