Universität Bremen

Praktische Informatik 1
Fehlerbehandlung 1

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

Universität Bremen

```
RingBuffer r = new RingBuffer(5);
int antimatter = r.pop();
```

Gründe für logische Fehler

- Fehlerhafte Implementierung (Entspricht nicht der Spezifikation)
- Ungültige Objektanfrage, z.B. Aufruf in unerlaubter Situation
- Inkonsistenter Objektzustand, z.B ausgelöst durch Klassenerweiterung

```
if (aktuellerRaum.gibAusgang(richtung) != null) {
    aktuellerRaum.gibAusgang(richtung)
            .ergänze(this);
    aktuellerRaum.entferne(this);
    aktuellerRaum = aktuellerRaum
            .gibAusgang(richtung);
}
```

```
class Teleporterraum extends Raum ...
    Raum gibAusgang(final String richtung)
    {
        return räume[zufall.nextInt(räume.length)];
    }
```

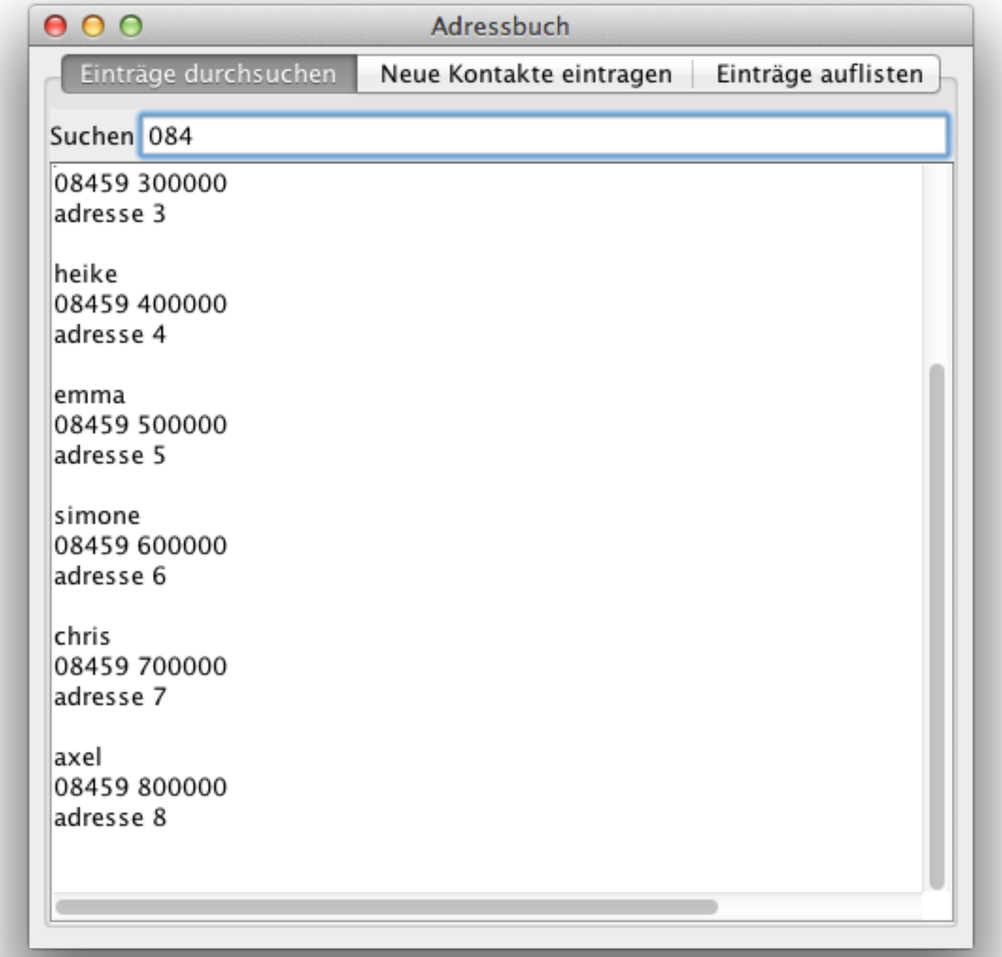
Universität Bremen

Fehler

- Fehler oft umgebungsbedingt (z.B. fehlerhafte Eingabe, Netzwerkunterbrechung)
- Dateiverarbeitung besonders anfällig (fehlende Dateien, fehlende Zugriffsrechte)
- Zwei Aspekte
  - Fehler melden
  - Fehler behandeln

Universität Bremen

Adressbuch: Demo



Universität Bremen

Defensive Programmierung

- Klient-Dienstleister
  - **Dienstleister**: bietet Funktionalität an
  - **Klient**: verwendet Funktionalität
- Verschiedene Sichtweisen
  - Klienten verhalten sich korrekt
  - Klienten machen Fehler, die verhindert werden müssen

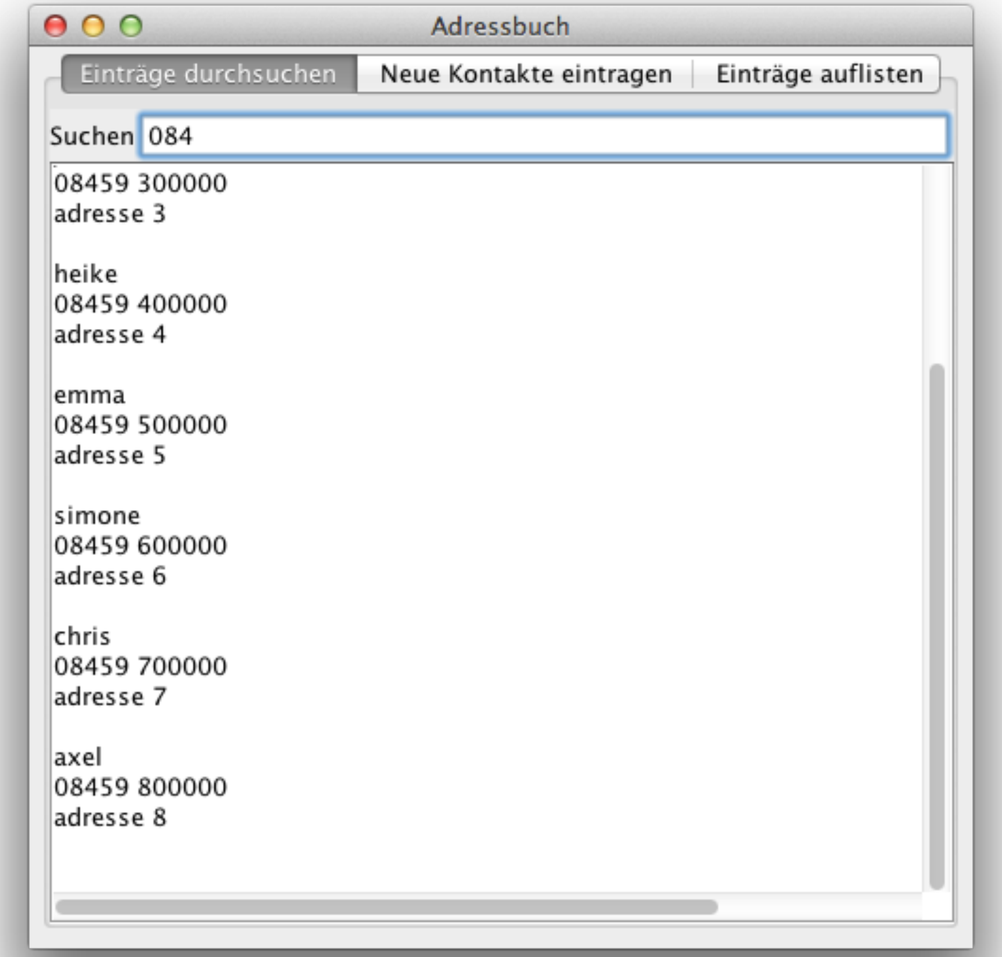
Universität Bremen

Aufgaben von Dienstleister und Klient

- Wie weit sollte ein Dienstleister die Anfragen von Klienten überprüfen?
- Wie sollte ein Dienstleister Fehler an seine Klienten melden?
- Wie kann ein Klient mögliche Scheiterungsgründe für eine Anfrage vorhersehen?
- Wie sollte ein Klient mit dem Scheitern einer Anfrage umgehen?

Universität Bremen

Parameter prüfen: Demo



Parameter prüfen

- Parameter machen Objekte **verwundbar**!
  - Parameter von Konstruktoren initialisieren das Objekt
  - Parameter von Methoden bestimmen das Verhalten
- Defensive Maßnahme: **Parameter überprüfen**

bedingung **?** *wenn wahr* **:** *wenn falsch*

```
Kontakt(final String n,
        final String t, final String a)
{
    name = n == null ? "" : n.trim();
    telefon = t == null ? "" : t.trim();
    adresse = a == null ? "" : a.trim();
}
```

```
public boolean equals(final Object jenes)
{
    return jenes instanceof Kontakt k
            && name.equals(k.gibName())
            && telefon.equals(k.gibTelefon())
            && adresse.equals(k.gibAdresse());
}
```

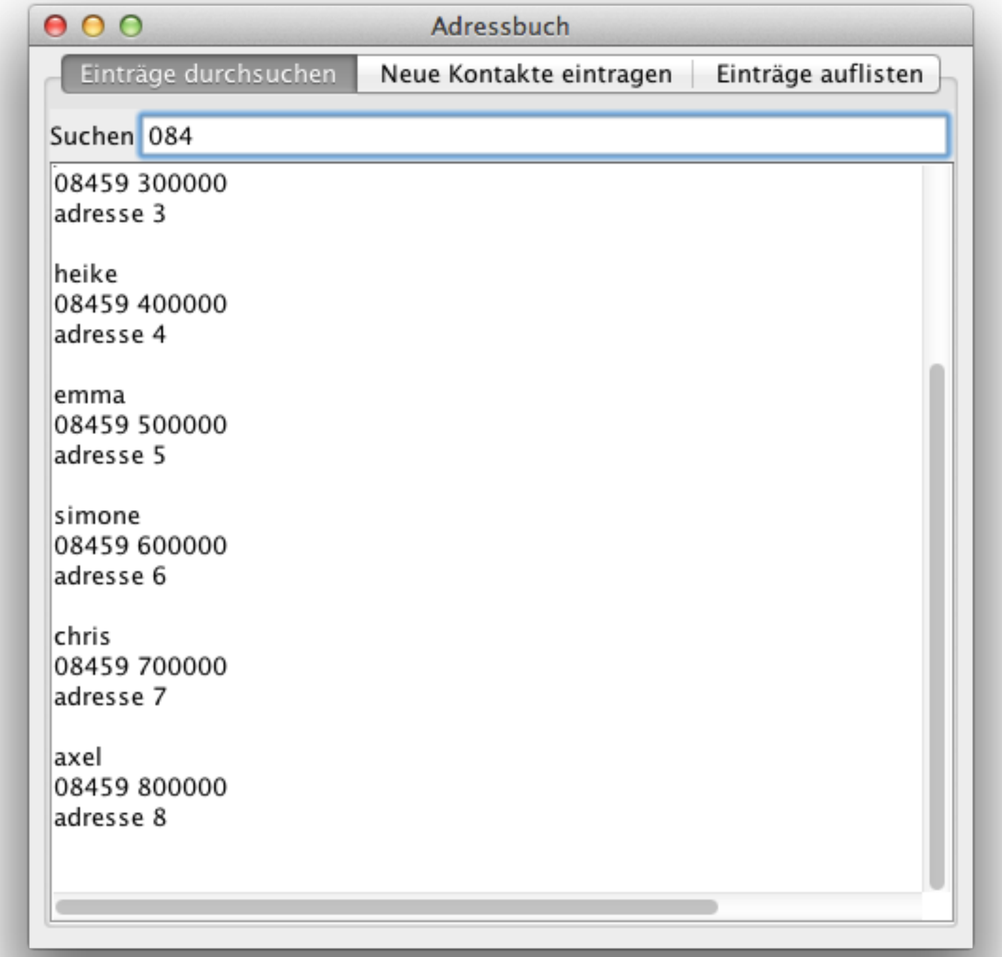
Universität Bremen

Fehlermeldungen durch den Dienstleister

- Fehler nur zu unterdrücken ist falsch, da sie auf Programmierfehler im Klienten hinweisen können
- Benutzer:in informieren?
  - Gibt es überhaupt Benutzer:in?
  - Können Benutzer:innen das Problem lösen?
- Klienten informieren?
  - Diagnosewert zurückliefern oder **Exception** werfen

Universität Bremen

Fehlerwert zurückgeben: Demo



Universität Bremen

Fehlerwert zurückgeben

- Als Fehlerwerte eignen sich Werte, die sich von regulären Ergebnissen unterscheiden (z.B. **null**, negative Zahlen bei Array-Index usw.)
- Klient kann Rückgabewert auswerten
  - Versuchen, Fehler zu beheben
  - Programmversagen vermeiden
- Klient kann Rückgabewert ignorieren
  - Kann nicht verhindert werden und führt wahrscheinlich zu Programmversagen

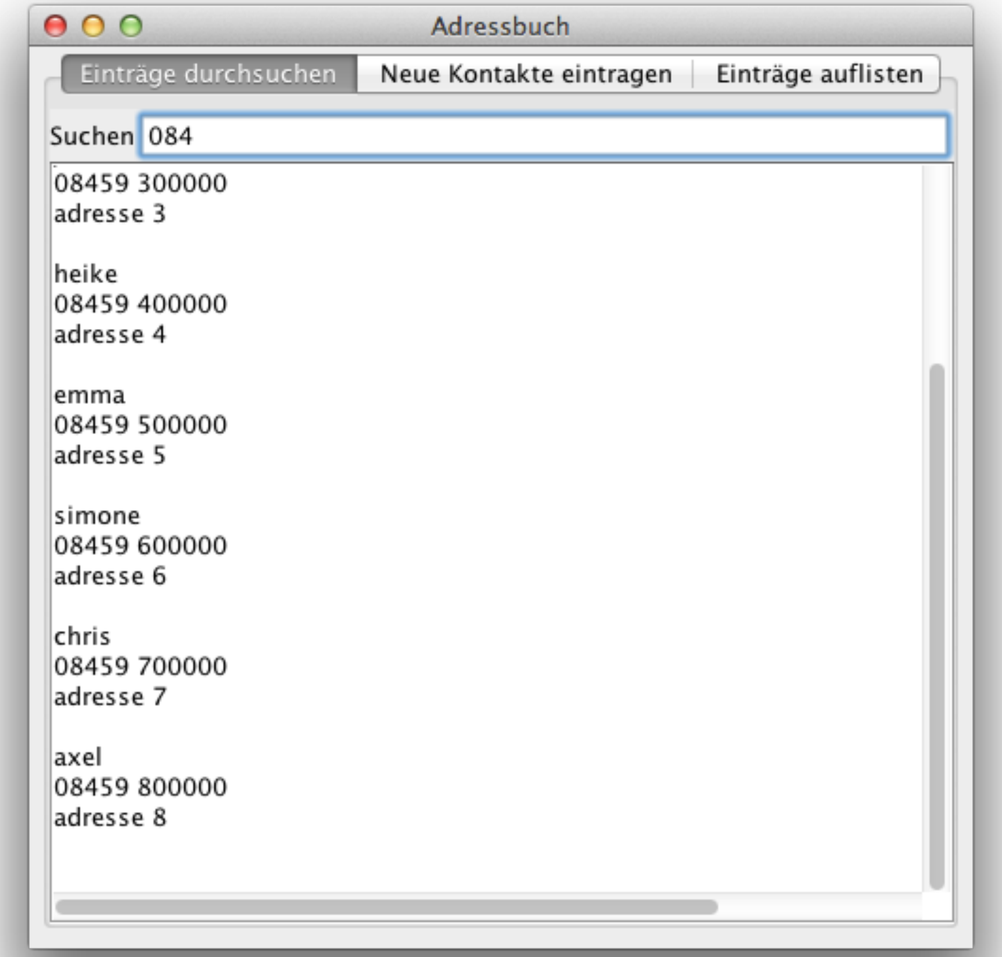
Universität Bremen

Ausnahmebehandlung

- Eine **Exception** (Ausnahme) ist ein Objekt, das Informationen über einen Programmfehler enthält
- Sie wird ausgelöst, um zu signalisieren, dass ein Fehler aufgetreten ist
- Exceptions können nicht (passiv) ignoriert werden
  - Normaler Kontrollfluss wird unterbrochen
- Eine unbehandelte Exception bricht das Programm ab

Universität Bremen

Ausnahmen erzeugen: Demo



Universität Bremen

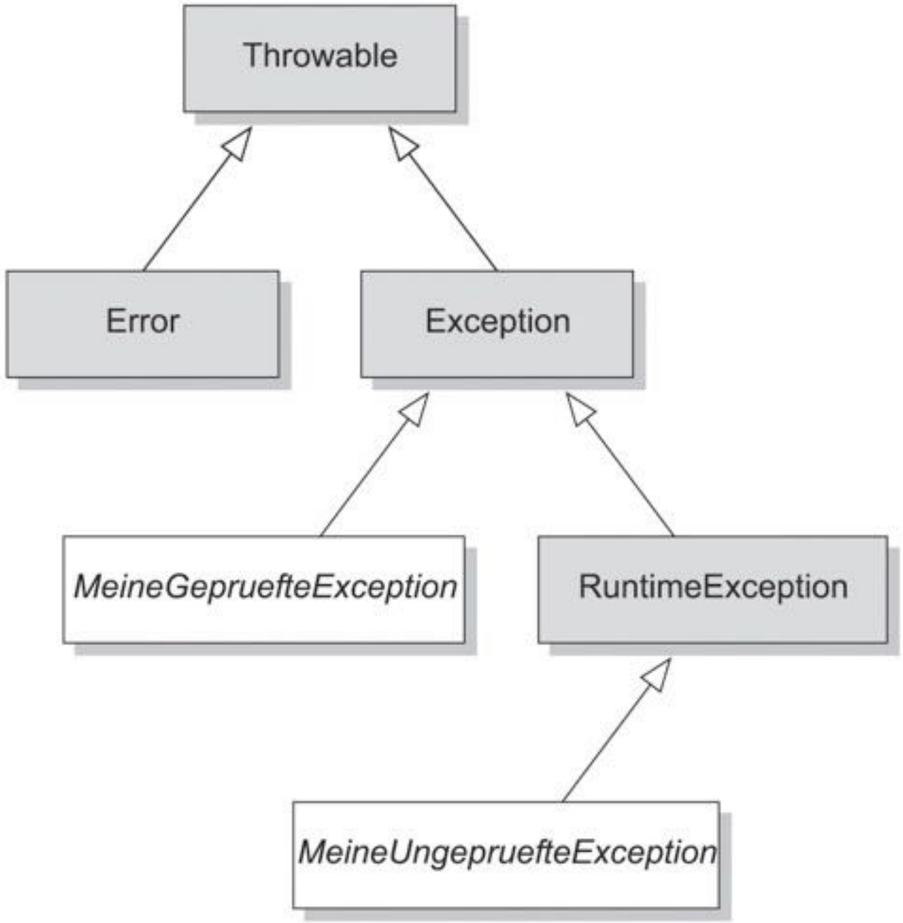
Auslösen einer Exception

- Eine Exception wird mit einer **throw**-Anweisung geworfen

```
 * @throws IllegalArgumentException Wenn schlüssel null ist.
 */
Kontakt gibKontakt(final String schlüssel)
{
    if (schlüssel == null) {
        throw new IllegalArgumentException(
                "Parameter in gibKontakt ist null.");
    }
    return buch.get(schlüssel);
}
```

Exception-Klassen

- **Geprüfte Exceptions**: Subklassen von **Exception**
  - Müssen behandelt werden
  - Verwendung bei vorhersehbaren Fehlern
  - Fehlerbehandlung üblicherweise möglich
- **Ungeprüfte Exceptions**: Subklassen von **RuntimeException**
  - Müssen nicht behandelt werden
  - Verwendung bei unvorhersehbaren Fehlern
  - Fehlerbehandlung oft schwierig



Universität Bremen

Auswirkungen einer Exception

- Auslösende Methode wird vorzeitig beendet
- Sie hat **keinen** Rückgabewert
- Kontrolle kehrt **nicht** zum Aufrufpunkt des Klienten zurück
  - Klient kann nicht einfach fortfahren
- Klient kann Exception aber **fangen**

```
if (schlüssel == null) {
   throw new IllegalArgumentException(
            "Parameter in gibKontakt ist null.");
   // Kein return
}
else {
   return buch.get(schlüssel);
}
```

Universität Bremen

Ungeprüfte Exceptions

- Einsatz wird nicht vom Compiler überprüft
  - Z.B. keine Auswirkung auf Methodensignatur
- Ohne Fangen erfolgt Programmabbruch
  - Gängige Praxis

```
Kontakt gibKontakt(final String schlüssel)
{
    if (schlüssel == null) {
        throw new IllegalArgumentException(
                "Parameter in gibKontakt ist null.");
    }
    else if (schlüssel.trim().length() == 0) {
        throw new IllegalArgumentException(
                "Leerer Schlüssel in gibKontakt.");
    }
    return buch.get(schlüssel);
}
```

- Ungeprüfte Exceptions sollten verhindert und nicht gefangen werden!

Universität Bremen

Objekterzeugung verhindern

- Konstruktoren können keinen (Fehler-)Wert zurückgeben
- Sie können aber durch Werfen einer Exception die Erzeugung ihres Objekts verhindern

```
Kontakt(final String name, final String telefon, final String adresse)
{
    this.name = name == null ? "" : name.trim();
    this.telefon = telefon == null ? "" : telefon.trim();
    this.adresse = adresse == null ? "" : adresse.trim();
    if (this.name.isEmpty() && this.telefon.isEmpty()) {
        throw new IllegalArgumentException(
          "name und telefon dürfen nicht beide leer sein.");
    }
}
```

Universität Bremen

Behandlung von Ausnahmen

```
import java.io.IOException;
:
void speichere(final String datei)
    throws IOException
:
```

- Methoden, die **geprüfte Exceptions** werfen, müssen diese in ihrer Signatur hinter **throws** angeben
- Methoden, die Methoden mit **throws**-Klausel aufrufen, müssen
  - die genannten Exceptions fangen oder
  - sie selbst in ihrer Signatur nennen
- Dadurch entsteht eine Verpflichtung zur Fehlerbehandlung

Universität Bremen

Exceptions fangen

- **Exception-Handler**: Ein Programmabschnitt, der Anweisungen schützt, in denen eine Exception ausgelöst werden kann

- Definiert Anweisungen zur Meldung und/oder zum Wiederaufsetzen nach einem Fehler

```
String datei = null;
try {
    datei = Benutzer fragen
    adressbuch.speichere(datei);
}
catch (final IOException e) {
    System.out.println("Speichern in "
            + datei + " schlug fehl.");
}
```

Exception wird ausgelöst

Ausführung wird fortgesetzt

```
try {
    eine oder mehrere geschützte Anweisungen
}
catch (ExceptionTyp e) {
    Exception melden und eventuell wieder aufsetzen
}
```

Universität Bremen

Werfen und Fangen mehrerer Exceptions

```
try {
    ...
    ref.verarbeite();
    ...
}
catch (final EOFException e) {
    // angemessene Behandlung einer Dateiende-Exception (end-of-file)
    ...
}
catch (final FileNotFoundException e) {
    // angemessene Behandlung für eine nicht gefundene Datei
    ...
}
```

```
void verarbeite() throws EOFException, FileNotFoundException
...
```

Universität Bremen

Fangen mehrerer Exceptions

- Mehrere Exceptions können auch mit einem (multi-)**catch** gefangen werden
- Mehrere Exception-Typen werden durch **|** voneinander getrennt und dürfen nicht voneinander erben
- Statischer Typ der Referenz ist dichteste gemeinsame Superklasse
- Referenz ist implizit **final**

```
try {
   ref.verarbeite();
}
catch (EOFException | FileNotFoundException e) {
    // angemessene Behandlung beider Exceptions
    ...
}
```

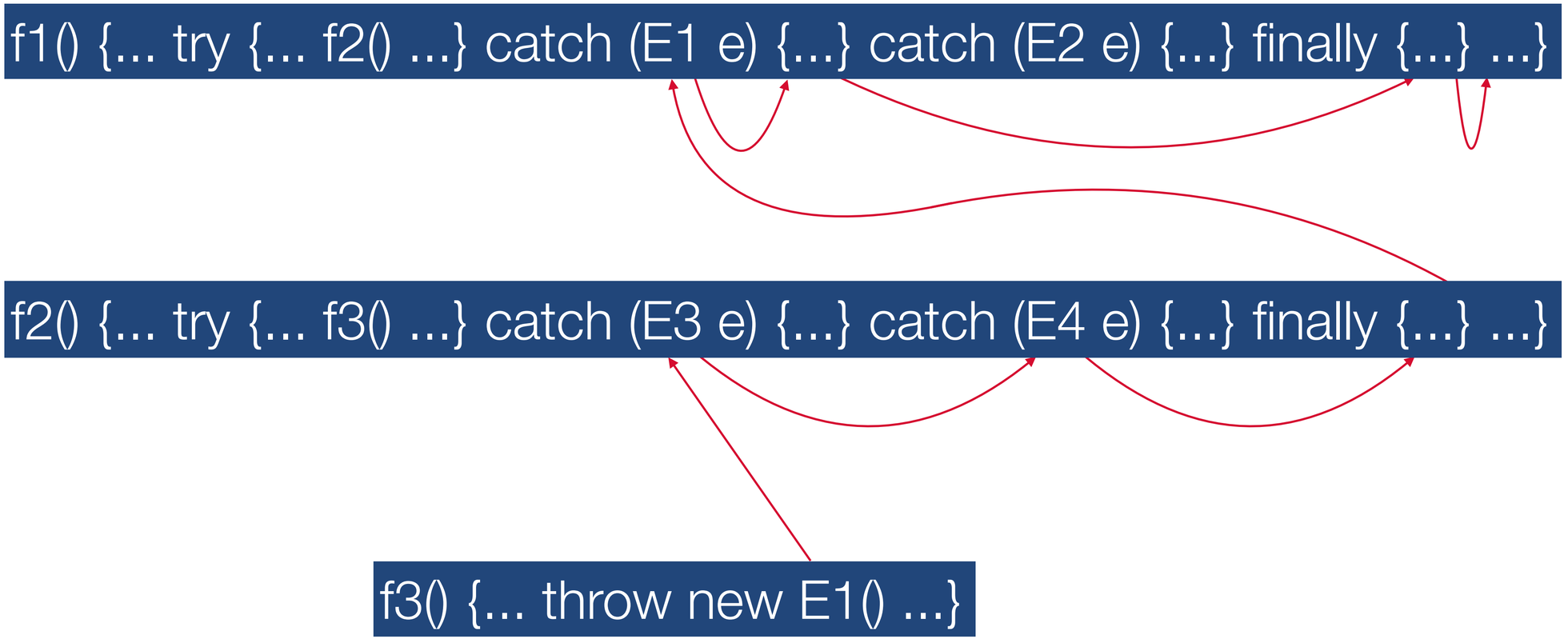
Universität Bremen

finally

- Wird **immer** ausgeführt, auch wenn
  - eine **return**-Anweisung in **try** oder **catch** steht
  - in **try** eine Exception ausgelöst, aber nicht gefangen wurde

```
try {
    eine oder mehrere
    geschützte Anweisungen
}
catch (Exception e) {
    Exception melden und
    eventuell wieder aufsetzen
}
finally {
    gemeinsame Aktionen, die
    ausgeführt werden müssen,
    unabhängig davon, ob eine
    Exception aufgetreten ist
    oder nicht
}
```

Der Weg einer Exception



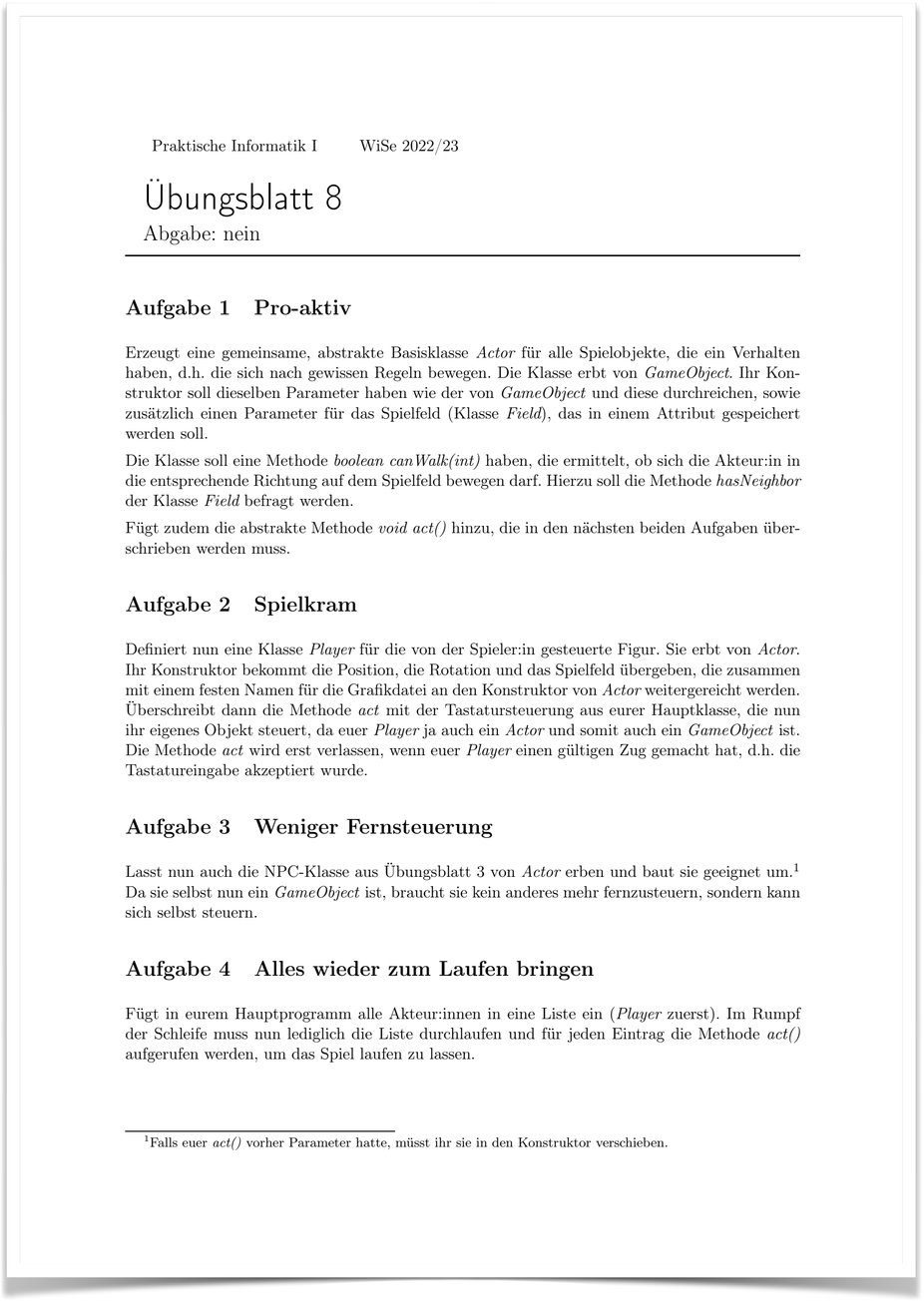
Universität Bremen

Zusammenfassung der Konzepte

- **Klient** und **Dienstleister**
- **Exception**
- **Ungeprüfte** und **geprüfte Exception**
- **Exception-Handler**

Übungsblatt 8

- Refactoring!
- Aufgabe 1: Abstrakte Basisklasse für Akteure
  - Erbt von GameObject
- Aufgabe 2: Eigene Klasse für Spielfigur, die von Basisklasse erbt
- Aufgabe 3: NPC-Klassen erben ebenfalls davon
- Aufgabe 4: Alle Akteure in Liste



Praktische Informatik I WiSe 2022/23

Übungsblatt 8

Abgabe: nein

Aufgabe 1 Pro-aktiv

Erzeugt eine gemeinsame, abstrakte Basisklasse *Actor* für alle Spielobjekte, die ein Verhalten haben, d.h. die sich nach gewissen Regeln bewegen. Die Klasse erbt von *GameObject*. Ihr Konstruktor soll dieselben Parameter haben wie der von *GameObject* und diese durchreichen, sowie zusätzlich einen Parameter für das Spielfeld (Klasse *Field*), das in einem Attribut gespeichert werden soll.

Die Klasse soll eine Methode *boolean canWalk(int)* haben, die ermittelt, ob sich die Akteur:in in die entsprechende Richtung auf dem Spielfeld bewegen darf. Hierzu soll die Methode *hasNeighbor* der Klasse *Field* befragt werden.

Fügt zudem die abstrakte Methode *void act()* hinzu, die in den nächsten beiden Aufgaben überschrieben werden muss.

Aufgabe 2 Spielkram

Definiert nun eine Klasse *Player* für die von der Spieler:in gesteuerte Figur. Sie erbt von *Actor*. Ihr Konstruktor bekommt die Position, die Rotation und das Spielfeld übergeben, die zusammen mit einem festen Namen für die Grafikdatei an den Konstruktor von *Actor* weitergereicht werden. Überschreibt dann die Methode *act* mit der Tastatursteuerung aus eurer Hauptklasse, die nun ihr eigenes Objekt steuert, da euer *Player* ja auch ein *Actor* und somit auch ein *GameObject* ist. Die Methode *act* wird erst verlassen, wenn euer *Player* einen gültigen Zug gemacht hat, d.h. die Tastatureingabe akzeptiert wurde.

Aufgabe 3 Weniger Fernsteuerung

Lasst nun auch die NPC-Klasse aus Übungsblatt 3 von *Actor* erben und baut sie geeignet um.[1] Da sie selbst nun ein *GameObject* ist, braucht sie kein anderes mehr fernzusteuern, sondern kann sich selbst steuern.

Aufgabe 4 Alles wieder zum Laufen bringen

Fügt in eurem Hauptprogramm alle Akteur:innen in eine Liste ein (*Player* zuerst). Im Rumpf der Schleife muss nun lediglich die Liste durchlaufen und für jeden Eintrag die Methode *act()* aufgerufen werden, um das Spiel laufen zu lassen.

[1] Falls euer *act()* vorher Parameter hatte, müsst ihr sie in den Konstruktor verschieben.