DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

Vorlesung Programmieren
Thema 10: Fehler, Ausnahmen und Zusicherungen

Olaf Herden

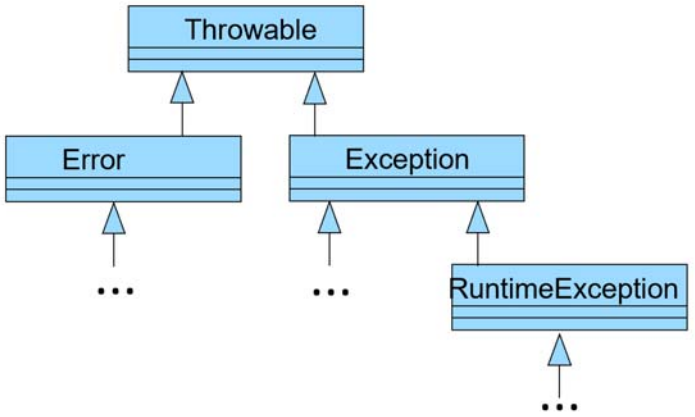
Fakultät Technik

Studiengang Informatik


```
( 1) public class Weiter{
( 2)   public static void main(String[] args){
( 3)     ...
( 4)     try{
( 5)       ...
( 6)     }
( 7)     catch (<Ausnahmetyp_1> e){
( 8)       <Behandlung von Ausnahmetyp_1>
( 9)     }
(10)     catch (<Ausnahmetyp_2> e){
(11)       <Behandlung von Ausnahmetyp_2>
(12)     }
(13)     finally{
(14)       <Code>
(15)     }
(16)     // Ab hier: Ganz normale Weiterverarbeitung
(17)     ...
(18)   }
(19) }
```

throws try

catch



Stand: 01/2024

Gliederung

- Arten von Fehlern
- Laufzeitfehler in Java (Ausnahmen)
- Ausnahmebehandlung
- Benutzerdefinierte Ausnahmen
- Design by Contract und Zusicherungen

Logische Fehler

- „Programm verhält sich nicht so, wie es soll“
- Bsp.: Ausführung Addition anstelle Multiplikation
- Ursachen:
  - Inkorrekte Systemanforderungen
  - Inkorrekte Umsetzung von Systemanforderungen in der Analysephase
  - Inkorrekte Umsetzung von korrekter Anforderung in Entwurfsphase
  - Inkorrekte Umsetzung von korrektem Systementwurf in Codierungsphase
- Behebung:
  - Testen (oder erst produktiver Einsatz) deckt Fehler auf
  - Korrektur des Quellcodes

Syntaktische Fehler

- „Programm entspricht nicht der Syntax der eingesetzten Programmiersprache“
- Bsp.: `class A extends A{...}`
- Ursache: Fehler beim Kodieren
- Behebung:
  - Compiler meldet Fehler
  - Korrektur des Quellcodes

Laufzeitfehler

- „Programm konnte korrekt kompiliert werden und verhält sich (zumindest meistens) so, wie es soll“
- Dennoch „gelegentlich“: Abbruch/Fehlerfall des Programms zur Laufzeit
- Bsp.:
  - Division durch 0
  - Index-Verletzung beim Array-Zugriff
- Ursache:
  - In Codierungsphase nicht alle möglichen Fälle bedacht
  - Falsche Programmeingaben

- Diese Klasse von Fehlern lässt sich nicht beheben
- Aber: Java (und einige andere Sprachen) bieten Exceptions (Ausnahmen) als Mechanismus zum Abfangen und zur Behandlung dieser Fehlerkategorie an

Übersicht

- Arten von Fehlern
- Laufzeitfehler in Java (Ausnahmen)
- Ausnahmebehandlung
- Benutzerdefinierte Ausnahmen
- Design by Contract und Zusicherungen

Beispiel (I)

- Gegeben sei folgendes Programm:

```
(1) public class Ausnahme{
(2)   public static void main(String[] args){
(3)     long i = Long.parseLong(args[0]);
(4)     System.out.println ("Das Quadrat von "+i+" ist "+i*i+".");
(5)   }
(6) }
```

- Funktion:
  - Über Kommandozeile Eingabe einer Zahl
  - Berechnung und Ausgeben des Quadrats der Zahl
- Beim Übergeben einer Zahl verläuft alles wie erwartet:

```
>java Ausnahme 5
Das Quadrat von 5 ist 25.
```

Beispiel (II)

- Kein Argument oder Buchstabe als Parameter ⇒ Laufzeitfehler

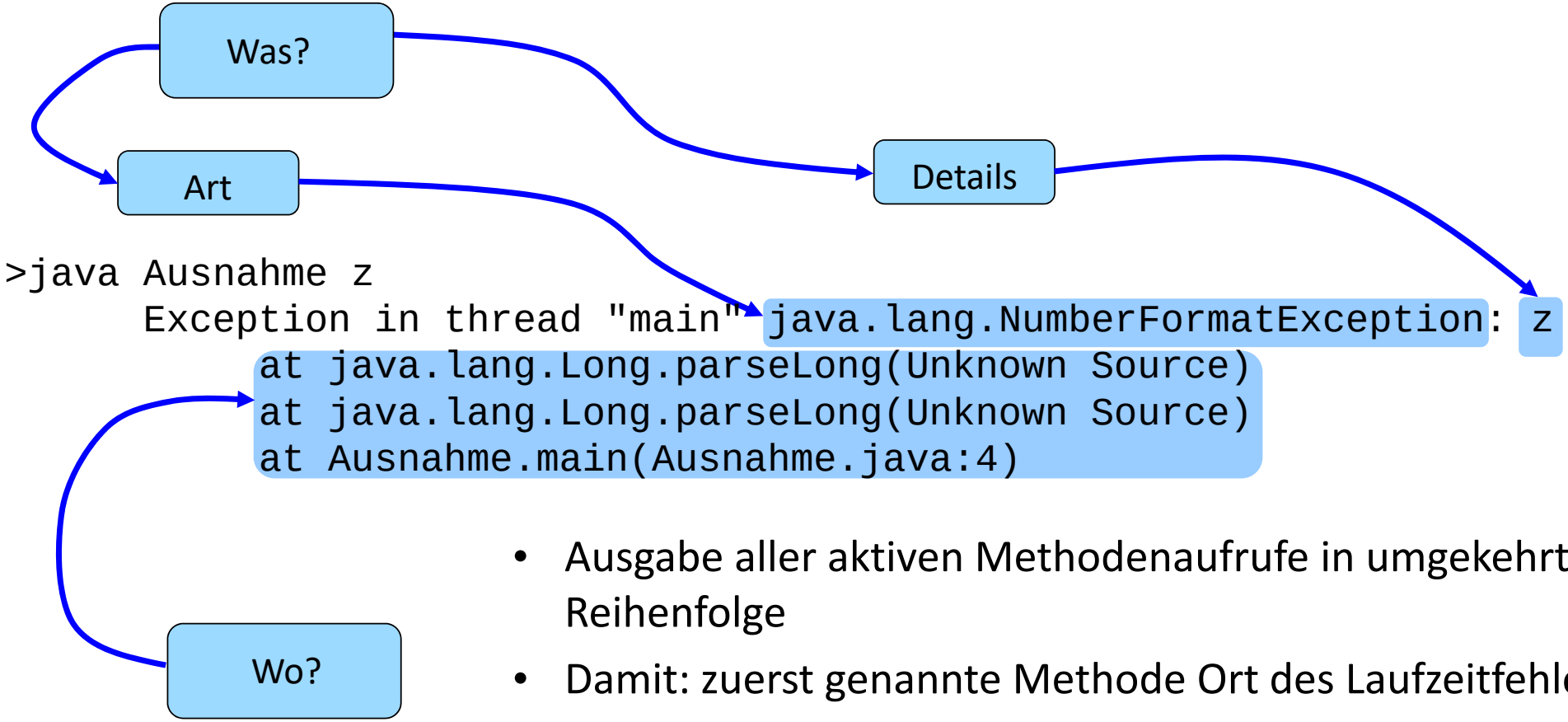
```
>java Ausnahme
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at Ausnahme.main(Ausnahme.java:4)
```

bzw.

```
>java Ausnahme z
Exception in thread "main" java.lang.NumberFormatException: z
     at java.lang.Long.parseLong(Unknown Source)
     at Ausnahme.main(Ausnahme.java:4)
```

- Man sagt auch: „Eine Ausnahme (Exception) wird erzeugt (geworfen).“

Ausnahme-Mitteilung

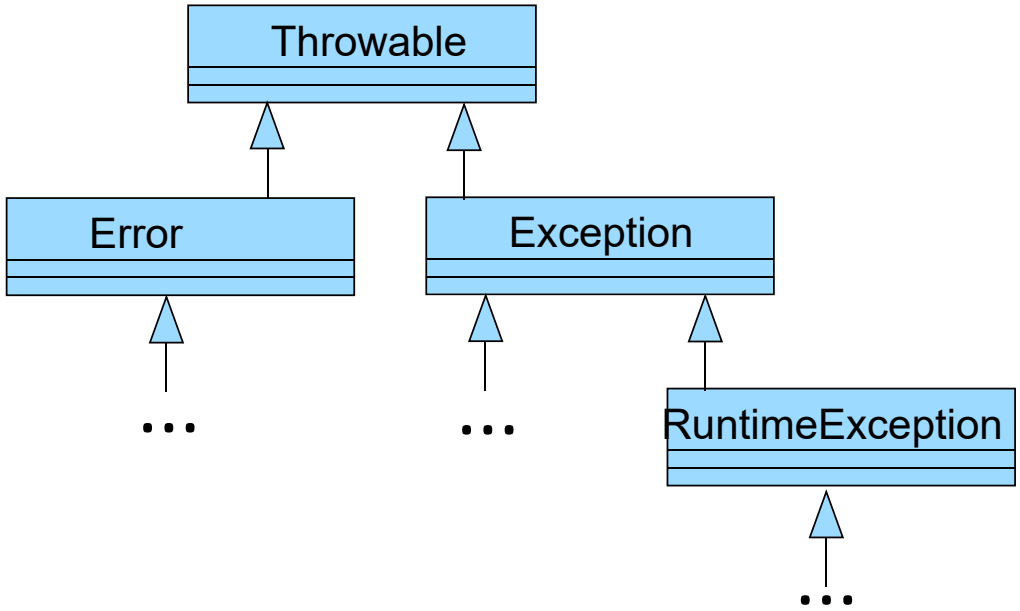


- Ausgabe aller aktiven Methodenaufrufe in umgekehrter Reihenfolge
- Damit: zuerst genannte Methode Ort des Laufzeitfehlers
- Bezeichnung Liste: „Stack Trace“

Technischer Ablauf

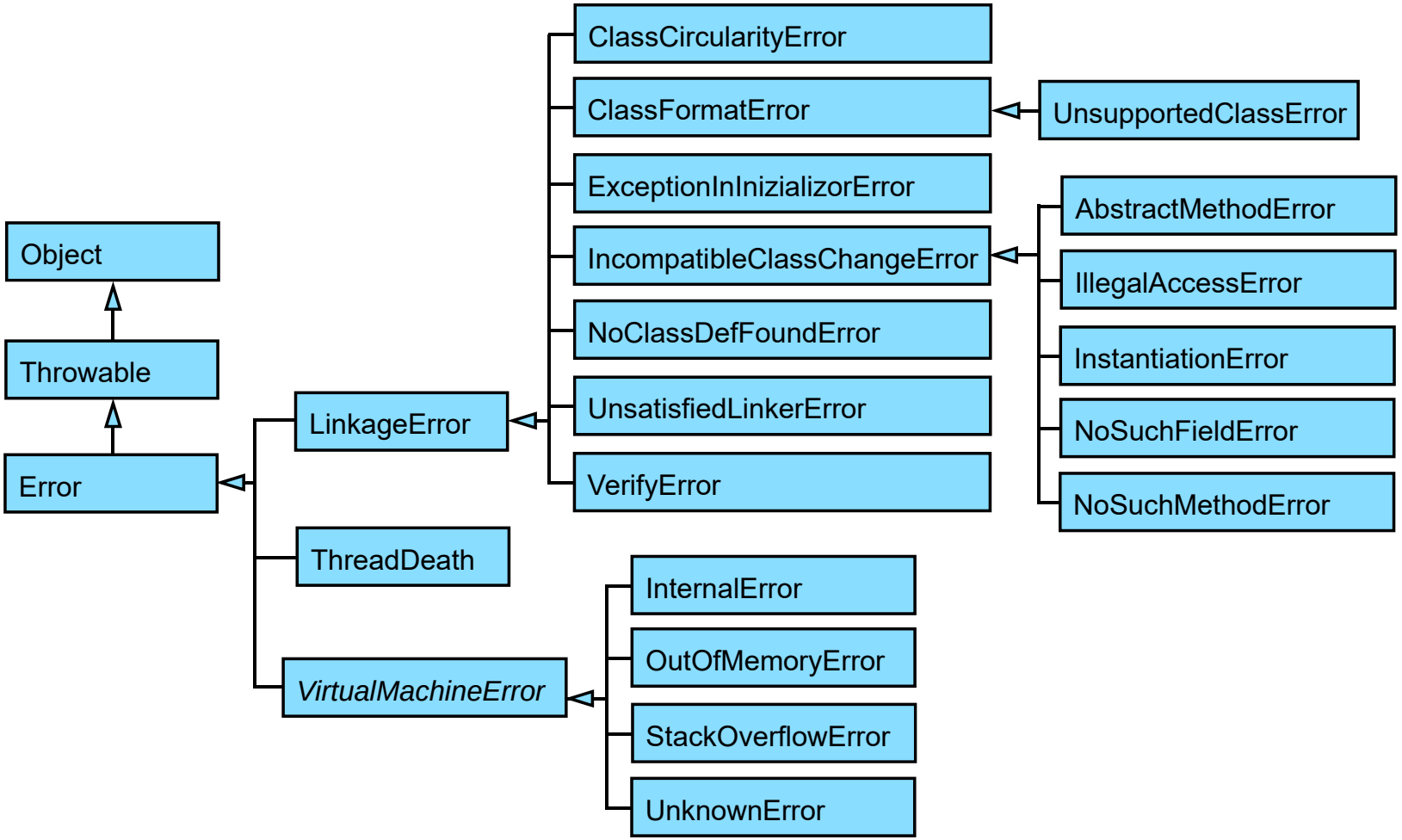
- JVM arbeitet Programm ab
- Beim Entdecken einer Ausnahme wird Objekt erzeugt
- Objekt Instanz einer Ausnahmeklasse
- Objekt enthält für Ausnahme relevante Informationen:
  - Art der Ausnahme
  - Details
  - Stack-Trace
- Paket `java.lang` definiert Reihe von Ausnahme- und Fehlerklassen

Ausnahmeklassen (I)

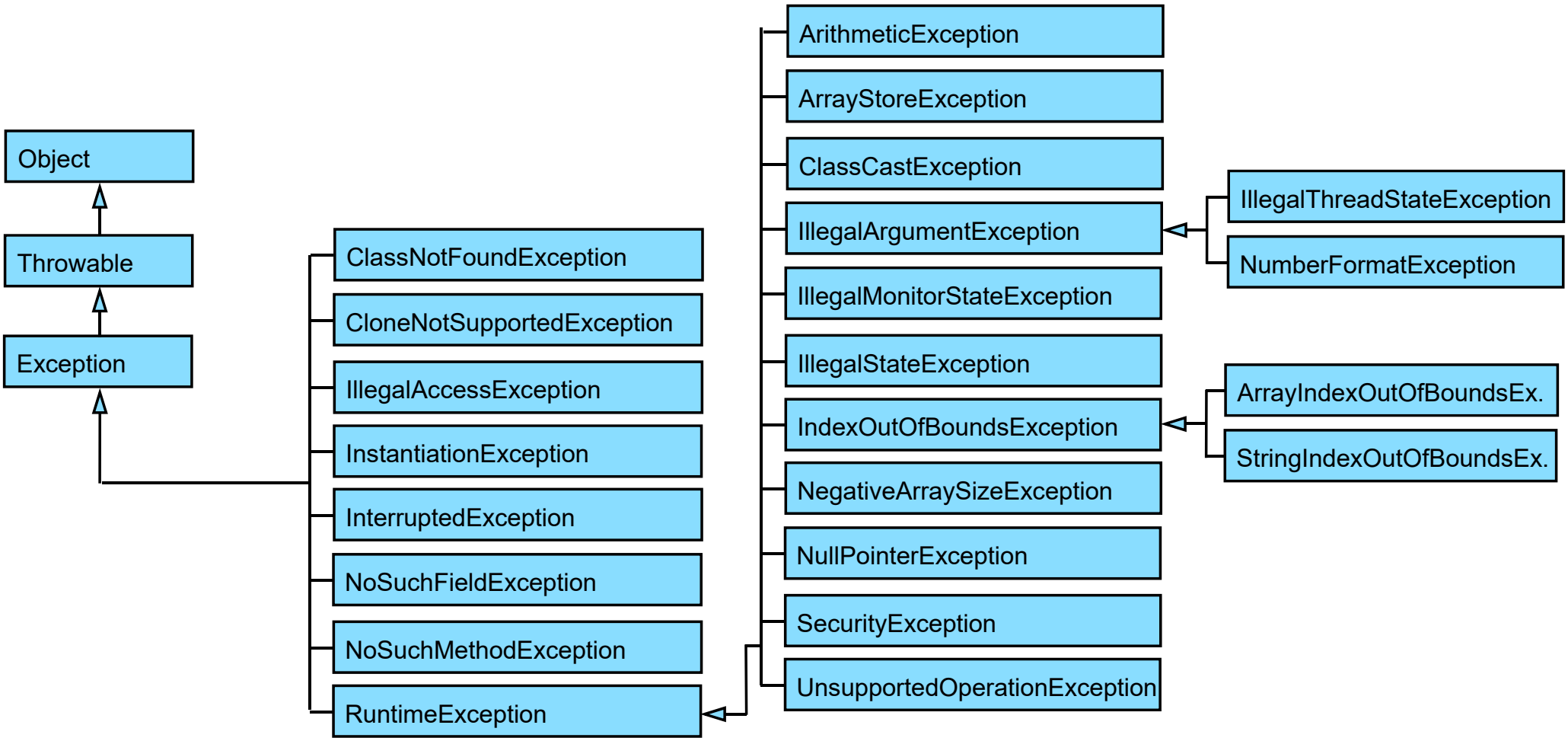


- `Throwable` gemeinsame Oberklasse aller Fehler und Ausnahmen
- Klassen im Zweig `Error` i.d.R. schwerwiegende Fehler, nicht zu behandeln
- Klassen im Zweig `Exception` behandelbare Ausnahmen
- Klassen im Zweig `RuntimeException` Systemfehler, keine Behandlung notwendig

Fehlerklassen im Paket `java.lang`



Ausnahmeklassen im Paket `java.lang`



Checked vs. Unchecked Exceptions

- Unchecked Exceptions:
  - Fehler im Programm (Division durch Null, Feldindex überschritten, …)
  - Unterklasse von `RuntimeException`
  - Methode muss keine Ausnahmebehandlung implementieren
- Checked Exceptions:
  - Fehler außerhalb des Kontrollflusses des aktuellen Programms (DB-Probleme, Netzwerkausfall, nicht vorhandene Dateien, …)
  - Unterklasse von `Exception`
  - Methode muss Ausnahmebehandlung implementieren

Übersicht

- Arten von Fehlern
- Laufzeitfehler in Java (Ausnahmen)
- Ausnahmebehandlung
- Benutzerdefinierte Ausnahmen
- Design by Contract und Zusicherungen

Definition

- Ausnahmebehandlung := Reaktion des Programms bei Auftreten einer Ausnahme

- Standardfall („Nichts tun“):
  - Ausgabe im Ausnahmeobjekt gespeicherter Informationen
  - Abbruch des Programms
- Benutzerdefinierte Ausnahmebehandlung: `try-catch`-Anweisung

try-catch-Anweisung (I)

- Syntax:

```
try {
   <Programmblock>
}
catch(<Ausnahmetyp_1> Ausnahmeobjektname_1){
    <Ausnahmebehandlung>
}
catch(<Ausnahmetyp_2> Ausnahmeobjektname_2){
   <Ausnahmebehandlung>
}
...
```

- Semantik: Bei Auftreten von Ausnahme im Programmblock, Sprung zum `catch`-Block des Ausnahmetyps, Ausführung hier definierter Ausnahmebehandlung

try-catch-Anweisung (II): Beispiel

- Beispiel von oben erweitert um `try-catch`-Anweisung:

```
( 1) public class TryCatch{
( 2)   public static void main(String[] args){
( 3)     try{
( 4)       long i = Long.parseLong(args[0]);
( 5)       System.out.println("Das Quadrat von "+i+" ist "+i*i+".");
( 6)     }
( 7)     catch (NumberFormatException e){
( 8)       System.out.println("Übergebener Wert muss eine Zahl sein!");
( 9)     }
(10)   }
(11) }
```

- Aufruf

  `>java TryCatch z`

  führt zur Ausgabe

  `Übergebener Wert muss eine ganze Zahl sein!`

try-catch-Anweisung (III): Abarbeitung

```
( 1) public class TryCatch{
( 2)   public static void main(String[] args){
( 3)     try{
( 4)       long i = Long.parseLong(args[0]);
( 5)       System.out.println("Das Quadrat von "+i+" ist "+i*i+".");
( 6)     }
( 7)     catch (NumberFormatException e){
( 8)       System.out.println("Übergebener Wert muss eine Zahl sein!");
( 9) }}}
```

An dieser Stelle tritt der Fehler auf.

Ausnahmeobjekt o vom Typ der Ausnahme (hier: `NumberFormatException`) wird von Laufzeitumgebung erzeugt.

Ist ein `try-catch`-Block definiert?
Wenn nein ⇒ Programm abbrechen, Inhalt von o auf Konsole ausgeben
Wenn ja: Entspricht Typ von o dem beim `catch` definierten Typen?
  Wenn nein ⇒ Programm abbrechen, Inhalt von o auf Konsole ausgeben
  Wenn ja ⇒ Objekt o an Variable e binden und Block hinter dem `catch` abarbeiten

Mehrere `catch`-Blöcke

- Zweiter Ausnahmefall (kein Argument übergeben): Kein Abfangen, d.h. Abbruch und Ausgabe der Ausnahmeinformationen auf Konsole
- Lösung: Definition mehrerer `catch`-Blöcke, jeder behandelt einen Ausnahmetyp
- Beispiel:

```
( 1) public class TryCatch{
( 2)   public static void main(String[] args){
( 3)     try{long i = Long.parseLong(args[0]);
( 4)       System.out.println
( 5)           ("Das Quadrat von "+i+" ist "+i*i+".");
( 6)     }
( 7)     catch (NumberFormatException e){
( 8)       System.out.println("Übergebener Wert muss eine ganze Zahl sein!");
( 9)     }
(10)     catch (ArrayIndexOutOfBoundsException e){
(11)       System.out.println(„Fehlender Parameter beim Aufruf des Programms !");
(12)     }
(13) }}}
```

Für jede zu behandelnde Ausnahme ein `catch`-Block

Sonstige Ausnahmen (I)

- Mit dieser Vorgehensweise kann eine beliebige Anzahl von Ausnahmetypen behandelt werden
- Alle anderen Ausnahmetypen enden immer noch mit Programmabbruch
- Diese sind kompatibel zum Supertyp `Throwable` und können auf diese Weise aufgefangen werden

Sonstige Ausnahmen (II)

- Beispiel:

```
( 1) public class TryCatch{
( 2)   public static void main(String[] args){
( 3)     try{long i = Long.parseLong(args[0]);
( 4)       System.out.println
( 5)           ("Das Quadrat von "+i+" ist "+i*i+".");
( 6)     }
( 7)     catch (NumberFormatException e){
( 8)        <Behandlung NumberFormatException-Ausnahme>}
( 9)     catch (ArrayIndexOutOfBoundsException e){
(10)       <Behandlung ArrayIndexOutOfBoundsException-Ausnahme>}
(11)     catch (Throwable e){
(12)       <Behandlung anderer Ausnahmen>}
(13)   }
(14) }
```

Hier werden alle zuvor nicht behandelten Ausnahmen behandelt.

Weiterverarbeitung nach Ausnahmen (I)

- Aufgetretene Ausnahme in `catch`-Block abgefangen, Code danach wird verarbeitet
- Beispiel:

```
( 1) public class Weiter{
( 2)   public static void main(String[] args){
( 3)     try{
( 4)       long i = Long.parseLong(args[0]);
( 5)       System.out.println("Das Quadrat von "+i+" ist "+i*i+".");
( 6)     }
( 7)     catch (NumberFormatException e){ ... }
( 8)     catch (ArrayIndexOutOfBoundsException e){ ... }
( 9)     catch (Throwable e){ ... }
(10)     // Ab hier: Ganz normale Weiterverarbeitung
(11)     int summe = 0;
(12)     for (int j=1; j<=10; j++){summe += j;}
(13)     System.out.println(summe);
(14)   }
(15) }
```

Egal, ob im oberen Bereich Ausnahme auftritt oder nicht, dieser Code wird immer ausgeführt.

Weiterverarbeitung nach Ausnahmen (II)

- Beispiel ohne Ausnahme:

  ```
  >java Weiter 5
  ```

  führt zur Ausgabe

  ```
  Das Quadrat von 5 ist 25.
  55
  ```

- Beispiel mit Ausnahme:

  ```
  >java Weiter
  ```

  führt zur Ausgabe

  ```
  Fehlender Parameter beim Aufruf des Programms!
  55
  ```

`finally`-Block (I)

- `finally`-Block:
  - Optionales Ende `try-catch`-Anweisung
  - Ausführung in jedem Szenario, d.h. bei keiner Ausnahme, Auftreten behandelter oder unbehandelter Ausnahme
- Anwendungsbeispiele:
  - Schließen von Dateien
  - Schließen von Datenbankverbindungen
  - Beenden noch offener Netzwerkverbindungen

finally-Block (II): Beispiel

```
( 1) public class Weiter{
( 2)   public static void main(String[] args){
( 3)     ...
( 4)     try{
( 5)       ...
( 6)     }
( 7)     catch (<Ausnahmetyp_1> e){
( 8)       <Behandlung von Ausnahmetyp_1>
( 9)     }
(10)     catch (<Ausnahmetyp_2> e){
(11)       <Behandlung von Ausnahmetyp_2>
(12)     }
(13)     finally{
(14)       <Code>
(15)     }
(16)     // Ab hier: Ganz normale Weiterverarbeitung
(17)     ...
(18)   }
(19) }
```

Ausführung Codeblock unabhängig vom Auftreten einer Ausnahme

Schachtelung (I)

- Schachtelung von `try-catch`-Anweisungen möglich:
  - Kein passender `catch`-Ausdruck gefunden ⇒ Sprung aus innerem `try-catch`-Block in umgebenden Block
  - Auch hier keine passender `catch`-Ausdruck ⇒ Sprung in `try-catch`-Block der aufrufenden Methode
- Beispiel:

```
( 1) ...
( 2) try{
( 3)   try{
( 4)     int i = Integer.parseInt(args[0]);
( 5)   }
( 6)   catch (NumberFormatException e){
( 7)     System.out.println("Innen " : + e);
( 8)   }
( 9) }
(10) catch(Throwable e){
(11)   System.out.println("Aussen " : + e);
(12) }
```

Auftreten Ausnahme
⇒ Sprung ins nächste `catch`

Ausnahmetyp nicht passend
⇒ Sprung in umgebendes `catch`

Wäre Ausnahmetyp nicht passend
⇒ Weiterleitung an aufrufende Methode

Schachtelung (II)

- Ausführen vorhandener `finally`-Block vor Weiterleiten
- Sofortige Beendigung der Bearbeitung aktueller Methode beim Weiterleiten des Ausnahmeobjektes an aufrufende Methode

Weiterleiten

- Verwendung von `throw`-Anweisung im `catch`-Block:
  - Wird als „Auffangen und Weiterwerfen“ bezeichnet
  - Sinn: Anhängen wichtiger Zusatzinformationen

- Angabe von `throws`-Anweisung im Methodenkopf:
  - Keine Behandlung der Ausnahme, sondern Weiterleitung an aufrufende Methode
  - Beispiel:

```
(1) void openFile(String s) throws FileNotFoundException{
(2)    ...
(3)    Öffne die Datei mit Namen s
(4) }
```

  - Bei Angabe ungültigen Dateinamens ⇒ Weiterleitung der erzeugten `FileNotFoundException`-Ausnahme an aufrufende Methode

Kein `catch`-Block

- Kein `catch`-Block zulässig, z.B.

```
( 1) {
( 2)   try{
( 3)     ...
( 4)   }
( 5)   finally{
( 6)     ...
( 7)   }
( 8) }
```

- Verwendung sinnvoll bei Weiterleitung von Ausnahme an aufrufende Methode
- Ausführung Code im `finally`-Block

Leerer `catch`-Block

- Leerer `catch`-Block zulässig, z.B.

```
( 1) {
( 2)   try{
( 3)     int i = Integer.parseInt(args[0]);
( 4)   }
( 5)   catch (NumberFormatException e){
( 6)     ...
( 7)   }
( 8) }
```

- Verwendung nicht sinnvoll
- Ausnahme wird unterdrückt

Wiederholung `try`-Block (I)

- Manchmal wünschenswert: Nach Auftreten Ausnahme Wiederholung `try`-Block (bis keine Ausnahme mehr auftritt)
- Beispiel: Benutzereingabe
- Keine Unterstützung in Java in Form einer Anweisung
- Folgender Workaround:
  - Setze gesamte `try-catch`-Anweisung in Endlosschleife
  - Verwende als letzte Anweisung in `try`-Block `break`-Anweisung

Wiederholung `try`-Block (II): Beispiel

```
( 1) import javax.swing.JOptionPane;
( 2)
( 3) public class WiederholeTry{
( 4)
( 5)   public static void main(String[] args){
( 6)     int inputNumber = 0;
( 7)     while(true){
( 8)       try{
( 9)         String s = JOptionPane.showInputDialog("Bitte Zahl eingeben:");
(10)         inputNumber = Integer.parseInt(s);
(11)         break;
(12)       }
(13)       catch(NumberFormatException e){
(14)         System.out.println("Das war keine Zahl!");
(15)       }
(16)     }
(17)     System.out.println("Die eingegebene Zahl war „ + inputNumber);
(18)   }
(19) }
```

Übersicht

- Arten von Fehlern
- Laufzeitfehler in Java (Ausnahmen)
- Ausnahmebehandlung
- Benutzerdefinierte Ausnahmen
- Design by Contract und Zusicherungen

Definition

- Bisher:
  - JVM erzeugt Ausnahme
  - Methoden aus Java API erzeugen Ausnahmen
- Wünschenswert:
  - Definition eigener Ausnahmetypen
  - Explizites Erzeugen dieser Ausnahmen von Objekten im eigenen Code
- Vorgehen:
  - Definieren eines eigenen Ausnahmetypen
  - Festlegen, welche Methode diese Ausnahme werfen darf
  - Programmieren des Anwendungscodes

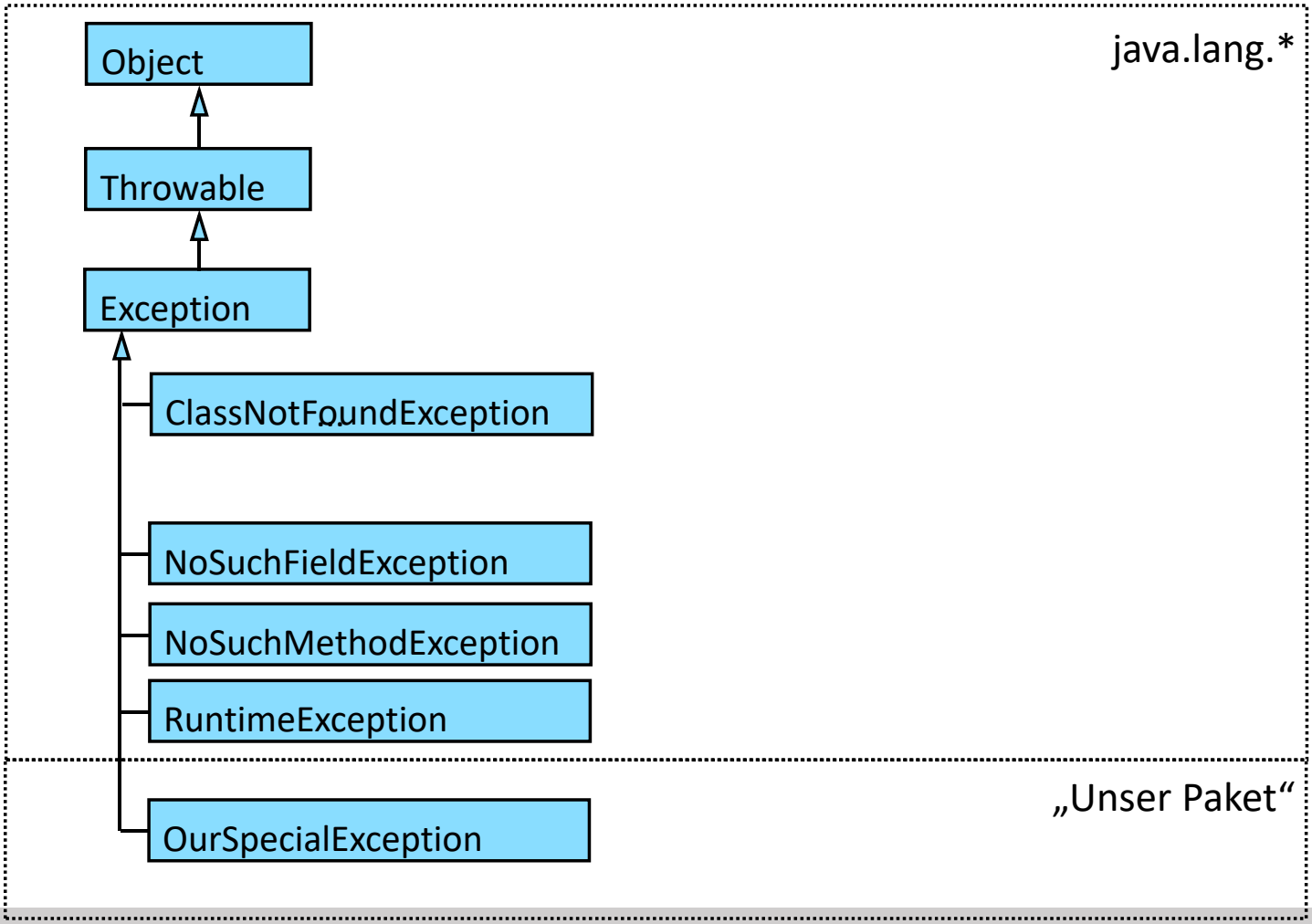
Definition eigener Ausnahmetyp (I)

- Anlegen Klasse für Ausnahmetyp, Ableiten von `Exception`
- Klasse `Exception` u.a. folgende zwei Methoden:
  - Konstruktor `public Exception(String s)`, erzeugt Ausnahmeobjekt mit angegebenen Detailinformationen
  - Methode `public String getMessage()`, gibt Detailinformationen der Ausnahme zurück
  - Methode `public printStackTrace()`, bewirkt Ausgabe des Stack Trace
- Beispiel:

```
(1) public class OurSpecialException extends Exception{
(2)   OurSpecialException(String s){
(3)     super(s);
(4)   }
(5) }
```

- Resultat: Erweiterung der Klassenbibliothek

Definition eigener Ausnahmetyp (II)



Definition Ausnahme werfende Methode (I)

- Methoden können nun Objekte neu definierter Ausnahme erzeugen
- Syntax: `<Modifier> <Rückgabetyp> Methodenname(Parameterliste)`
  `throws <Ausnahmetyp> { ... }`
- Explizites Erzeugen von Ausnahmeobjekten im Methodenrumpf:
  `throw new <Ausnahmetyp>(<Parameter>);`
- Beispiel:
  - Klasse mit Methode `berechne()`
  - `berechne()` hat Integer-Parameter i
  - Falls i positiv, gerade und kleiner oder gleich 10 $\Rightarrow$ Rückgabe i^2
  - Sonst: Erzeuge Ausnahmeobjekt vom Typ `OurSpecialException`

Definition Ausnahme werfende Methode (II)

```
( 1) public class Klasse{
( 2)   public static int berechne(int i)
( 3)     throws OurSpecialException{
( 4)      // Ausnahmen festlegen und Objekte erzeugen
( 5)      if (i<0){
( 6)        throw new OurSpecialException
( 7)                  ("Wert von i zu klein: "+i);}
( 8)      if ((i % 2) == 1){
( 9)        throw new OurSpecialException
(10)                  ("Ungerader Wert von i : "+i);}
(11)      if (i>10){
(12)        throw new OurSpecialException
(13)                  ("Wert von i zu groß: "+i);}
(14)      // Keine Ausnahme
(15)      return i*i;
(16)   }
(17) }
```

Erzeugung Ausnahmeobjekt für jeden Ausnahmefall

Anwendungsprogramm (I)

- Verwendung Methode im Anwendungsprogramm
- Beispiel:

```
( 1) public class Anwendung{
( 2)   public static void main (String[] args){
( 3)     try{
( 4)       System.out.println(Klasse.berechne(-5));
( 5)      }
( 6)     catch(OurSpecialException e){
( 7)       System.out.println("Es ist eine Ausnahme vom Typ
( 8)                          OurSpecialException aufgetreten.");
( 9)       System.out.println("Details : "+e.getMessage());
(10)     }
(11)   }
(12) }
```

- Unterschiedlichen Ausgaben bei verschiedenen Varianten im `try`-Block

Anwendungsprogramm (II): Aufruf (I)

- `System.out.println(Klasse.berechne(-5));`

  führt zu

  ```
  Es ist eine Ausnahme vom Typ OurSpecialException aufgetreten.
  Details : Wert von i zu klein: -5
  ```

- `System.out.println(Klasse.berechne(0));`

  führt zu

  ```
  0
  ```

- `System.out.println(Klasse.berechne(3));`

  führt zu

  ```
  Es ist eine Ausnahme vom Typ OurSpecialException aufgetreten.
  Details : Ungerader Wert von i : 3
  ```

Anwendungsprogramm (III): Aufruf (II)

- ```
  System.out.println(Klasse.berechne(4));
  ```

  führt zu

  ```
  16
  ```

- ```
  System.out.println(Klasse.berechne(18));
  ```

  führt zu

  ```
  Es ist eine Ausnahme vom Typ OurSpecialException aufgetreten.
  Details : Wert von i zu groß: 18
  ```

Übersicht

- Arten von Fehlern
- Laufzeitfehler in Java (Ausnahmen)
- Ausnahmebehandlung
- Benutzerdefinierte Ausnahmen
- Design by Contract und Zusicherungen

Design by Contract (I)

- Methodik zur Entwicklung korrekter Programme
- Idee:
  - Zerlege Programm in kleine Komponenten (Klassen, Methoden)
  - Kleine Komponenten werden zu größeren Komponenten zusammengesetzt
  - Sind kleine Komponenten jeweils korrekt, dann auch zusammengesetzte
- Dabei folgende Prinzipien verfolgen:
  - Jede Komponente:
    - hat klar definierte Aufgabe
    - stellt ihre Funktionalität über öffentliche Schnittstelle zur Verfügung
    - kann selbst andere Komponenten benutzen
  - Komponenten sind unabhängig voneinander:
    - Unabhängige Entwicklung und Überprüfung
    - Austauschbarkeit von Komponenten mit gleicher Schnittstelle
    - Interne Änderungen in einer Komponente beeinflussen nicht andere Komponenten

Design by Contract (II)

- Definitionen für Methoden:
  - Vorbedingungen: Annahmen an Argumente der Methode und Zustand der Klasse
  - Nachbedingungen: Am Ende der Methode geltende Eigenschaften
  - Effektbeschreibung: Aufgetretene Seiteneffekte der Methode (z.B. Ausnahmen, Ein-/Ausgabe, …)
- Definitionen für Klassen:
  - Klasseninvariante:
    - Eigenschaft, die zu jedem Zeitpunkt für Klasse gelten müssen
    - Häufigste Anwendung: Beziehungen über mehrere Attribute, die für alle Objekte einer Klasse gelten müssen

- Klassen schließen Verträge, in denen sie sich verpflichten die Vor-/Nachbedingungen und Invarianten einzuhalten

`assert`-Anweisung (I)

- In Java steht `assert`-Anweisung zur Verfügung
- Syntax: `assert <Boolescher Ausdruck> : <Fehlermeldung>`
- Semantik:
  - Falls Boolescher Ausdruck wahr ⇒ Fortsetzung Verarbeitung mit nächster Anweisung
  - Falls Boolescher Ausdruck falsch ⇒ Erzeugung `AssertionError`-Objekt
- Beispiel:
  - Berechnung Binomialkoeffizient

$$\binom{n}{k} = \prod_{i=1}^{k} \frac{n+1-i}{i}$$

Beispiel (I)

```
( 1) public static long binom(long n, long k){
( 2)   assert n>=k : "Fehler: n muss kleiner gleich k sein";
( 3)   long retValue = 0;
( 4)   if (k == 0){
( 5)     return 1;
( 6)   }
( 7)   else{
( 8)     retValue = n;
( 9)     for (int i=2; i<=k; i++){
(10)       retValue *= (n + 1 - i);
(11)       retValue /=  i;
(12)     }
(13)   }
(14)   return retValue;
(15) }
```

- Aufruf von `binom(7,5)` führt zur Ausgabe von 21
- Aufruf von `binom(7,9)` führt zur Ausgabe von 0
- Ursache: Aktivierung Assertion-Überprüfung nicht Standard

Beispiel (II)

- Aktivierung der Assertion-Überprüfung mit JVM-Option `–ea` (enable assertions)
- Nun führt Aufruf von binom(7,9) führt zur Ausgabe

```
Exception in thread "main" java.lang.AssertionError: Fehler: n muss
kleiner gleich k sein
at AssertionBeispiel.binom(AssertionBeispiel.java:6)
at AssertionBeispiel.main(AssertionBeispiel.java:36)
```

- Ablauf:
  - JVM überprüft Bedingung in Zeile (2)
  - Wird als falsch ausgewertet
  - Erzeugung `AssertionError`-Objekt, Übergabe definierter Fehlertext als Parameter

Vor-/Nachbedingungen und Assertions

- Gegeben sei Klasse K mit Methode m mit Vorbedingung VB und Nachbedingung NB
- Folgender Java-Code:

```
( 1) class K{
( 2)   ...
( 3)   void m(...){
( 4)     assert VB : <Fehlermeldung>
( 5)     ...
( 6)     // Verarbeitung der Methode
( 7)     ...
( 8)     assert NB : <Fehlermeldung>
( 9)   }
(10)   ...
(11) }
```

Klasseninvarianten und Assertions (I)

- Gegeben sei Klasse K mit Invariante I und Methoden m_1, …, m_n mit Vor- und Nachbedingungen
- Implementierung:
  - Konstruktoren haben Invariante als Nachbedingung
  - Alle Methoden haben Invariante als Vor- und Nachbedingung
  - Anm.: Bedeutet, dass Invarianten während Methodenabarbeitung temporär verletzt sein dürfen

Klasseninvarianten und Assertions (II)

```
( 1) class K{
( 2)   ...
( 3)   // Für jeden Konstruktor
( 4)   K(...){
( 5)     // Konstruktor initialisiert Objekt
( 6)     ...
( 7)     assert I : <Fehlermeldung>
( 8)   }
( 9)   // Für jede Methode mi
(10)   void mi(...){
(11)     assert (VB && I) : <Fehlermeldung>
(12)     ...
(13)     // Verarbeitung der Methode
(14)     ...
(15)     assert (NB && I) : <Fehlermeldung>
(16)   }
(17)   ...
(18) }
```

Welche Realisierung? (I)

- Bsp.: Sicherer Feldzugriff über Index
- Verschiedene Möglichkeiten der Überprüfung:

- Lösung 1: Bedingte Anweisung

```
(1) if (i>=0 && i<myField.length){
(2)   doSomethingWith myField[i];
(3) }
(4) else{
(5)   doErrorHandling;
(6) }
```

Welche Realisierung? (II)

- Lösung 2: Exception

```
(1) try{
(2)   doSomethingWith myField[i];
(3) }
(4) catch(ArrayIndexOutOfBoundsException e){
(5)   doErrorHandling;
(6) }
```

- Lösung 3: Assertion

```
(1) assert (i>=0 && i<myField.length) : <ErrorMessage>
(2) doSomethingWith myField[i];
```

Welche Realisierung? (III)

- Folgende Konventionen für Java:
  - Verwendung Assertions nur zum Testen
  - Konsequenz:
    - In öffentliche Methoden:
      - Keine Assertions für Vorbedingungen
      - Verwendung Assertions für Nachbedingungen
    - Verwendung Assertions in privaten Methoden

- Kritik:
  - Assertions geraten zum reinen Testinstrument
  - Abrücken vom eigentlichen Ziel der korrekten Entwicklung
  - Weiteres mögliches Problem: Auch Assertions können fehlerhaft sein

Zusammenfassung (I)

- Arten von Fehlern:
  - Logische Fehler
  - Syntaxfehler
  - Laufzeitfehler

- Laufzeitfehler in Java (Ausnahmen):
  - Ausnahmen
  - Ausnahmemitteilung
  - Fehler-/Ausnahmeklassen

Zusammenfassung (II)

- Ausnahmebehandlung:
  - `try-catch`-Anweisung
  - `finally`-Block
  - Schachtelung
  - Weiterleiten von Ausnahmen
  - Fehlender und leerer `catch`-Block
  - Wiederholung des `try`-Blocks

- Benutzerdefinierte Ausnahmen:
  - Definition benutzerdefinierter Ausnahmen
  - Einsatz benutzerdefinierter Ausnahmen

Zusammenfassung (III)

- Design by Contract und Zusicherungen:
  - Design by Contract
  - `assert`-Anweisung
  - Vor-/Nachbedingungen mit `assert`-Anweisung realisieren
  - Klasseninvarianten mit `assert`-Anweisung realisieren

Aufgaben (I)

- **Aufgabe 1**

  Entwerfen Sie ein Paket, das folgende Komponenten enthält:

  - Eine benutzerdefinierte Ausnahme `MatrixIncompatibleForAdditionException`
  - Eine benutzerdefinierte Ausnahme `MatrixIncompatibleForMultiplicationException`
  - Die Klasse Matrix von Übungsblatt 8
  - Diese Klasse soll so modifiziert werden, dass bei der Addition und Multiplikation im Falle inkompatibler Formate jeweils die passende Ausnahme geworfen wird

Schreiben Sie eine Klasse Anwendung, die die Funktionsweise der beiden benutzerdefinierten Ausnahmen demonstriert.

Aufgaben (II)

- **Aufgabe 2**

  Implementieren Sie eine Klasse `PositiveVector`, die einen Zahlenvektor aus nur positiven Werten verwaltet. Neben Konstruktoren, `set`- und `get`-Methoden sollten auch die Skalarmultiplikation sowie Addition, Subtraktion, und Skalarprodukt zweier Vektoren als Methoden zur Verfügung stehen.

  Überlegen Sie sich Klasseninvarianten, Vor- und Nachbedingungen der einzelnen Methoden! Realisieren Sie das Programm einmal durch bedingte Anweisungen, einmal durch Ausnahmen und einmal durch Zusicherungen.