# Programmieren in Java

## Vorlesung 10: Exceptions, IO, Serialization

Robert Jakob

Albert-Ludwigs-Universität Freiburg, Germany

SS 2013

# Inhalt

# Exceptions

# Motivation

```
1  public class Statistics {
2
3      public int average(int[] vals) {
4          int sum = 0;
5          for (Integer n : vals) {
6              sum += n;
7          }
8          return sum / vals.length;
9      }
10
11     public static void main(String[] args) {
12         new Statistics().average(new int[] {});
13     }
14 }
```

- Programm enthält einen Fehler
- Was passiert in so einem Falle in Java?

# Ausgabe bei Aufruf

Die JavaVM beendet sich mit einer Fehlermeldung und gibt folgendes aus:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Statistics.average(Statistics.java:8)
    at Statistics.main(Statistics.java:12)
```

# Ausgabe bei Aufruf

Fehlermeldung besteht aus 3 Teilen:

**Exception** : `java.lang.ArithmeticException`
Art des Fehlers

**Fehlertext** : `/ by zero`
Genauere textuelle Beschreibung des Fehlers

**Stacktrace** : Ort des Fehlers und Aufrufskette

```
at Statistics.average(Statistics.java:8)
at Statistics.main(Statistics.java:12)
```

`Klasse.methode(Dateiname:Zeilennummer)`

# Arten von Fehlern

Java unterscheidet 3 Arten von Fehlern:

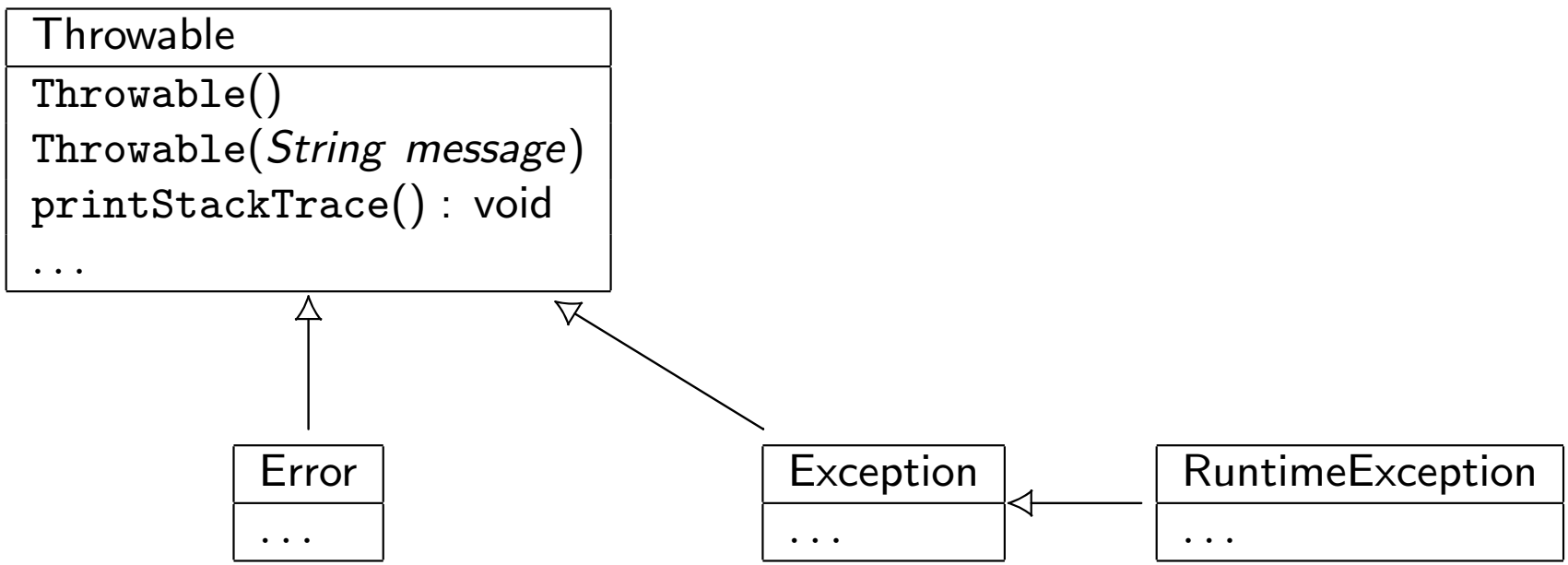
1. Ausnahmen, auf die man nur wenig Einfluss nehmen kann:
   - Kein Speicher mehr verfügbar
   - Stack overflow (rekursive Aufrufe)
2. Ausnahmen, die durch korrekte Programmierung vermeidbar sind:
   - Dereferenzierung von null
   - Arrayzugriff außerhalb der Grenzen
3. Ausnahmen, die nicht vermeidbar sind, aber behandelt werden können: Alternativer Rückgabewert
   - Datei kann nicht gelesen werden
   - Netzwerkverbindung ist gestört

Terminologie: Ausnahme vs. *Exception*

# Exceptions

In Java: Fehler repräsentiert durch Objekte

- Throwable ist Superklasse aller Fehler in Java
- Nicht vermeidbare Exceptions: Subklassen von Exception[1]
- Vermeidbare Exceptions: Subklassen von RuntimeException
- Error **extends** Throwable: z.B. OutOfMemoryError




[1] außer RuntimeException und Subklassen

# Werfen von Exceptions

Erzeugen/"werfen" einer Exception:

```
throw obj;
```

obj ist Instanz der Klasse Throwable oder einer ihrer Subklassen.

# Werfen von Exceptions

Erzeugen/"werfen" einer Exception:

```
throw obj;
```

obj ist Instanz der Klasse Throwable oder einer ihrer Subklassen.

Einfachste Form:

```
throw new Exception();
```

Konsequenzen:

- Unterbrechung der normalen Programmausführung
- JavaVM speichert "Stacktrace"
- JavaVM wird u.U. beendet

# Beispiel: Werfen von Exceptions

Welche Ausgabe kommt bei Ausführung folgenden Codes?

```
1 public class ExceptionExample {
2     public static void main(String[] args) {
3         throw new ArithmeticException("A message");
4     }
5 }
```

# Beispiel: Werfen von Exceptions

Welche Ausgabe kommt bei Ausführung folgenden Codes?

```
1 public class ExceptionExample {
2     public static void main(String[] args) {
3         throw new ArithmeticException("A message");
4     }
5 }
```

```
Exception in thread "main" java.lang.ArithmeticException: A message
        at ExceptionExample.main(ExceptionExample.java:3)
```

# Klassifizierung in Checked/Unchecked Exceptions

Art der Fehler:

- durch korrekte Programmierung vermeidbar
- unvermeidbar aber behandelbar

# Klassifizierung in Checked/Unchecked Exceptions

Art der Fehler:

- durch korrekte Programmierung vermeidbar
- unvermeidbar aber behandelbar

**Vermeidbare** Fehler werfen **unchecked** Exceptions

- Objekt nicht initialisiert (NullPointerException)
- Division durch Null (ArithmeticException)

Subklassen von `RuntimeException`

# Klassifizierung in Checked/Unchecked Exceptions

Art der Fehler:

- durch korrekte Programmierung vermeidbar
- unvermeidbar aber behandelbar

**Vermeidbare** Fehler werfen **unchecked** Exceptions

- Objekt nicht initialisiert (NullPointerException)
- Division durch Null (ArithmeticException)

Subklassen von `RuntimeException`

**Unvermeidbare** Fehler werfen **checked** Exceptions

- Datei wurde nicht gefunden (FileNotFoundException)
- Allgemeine Exception für Servlets (ServletException)

Subklassen von `Exception` mit Ausnahme von `RuntimeException`

# Checked/Unchecked Exceptions

- Checked Exceptions müssen in der Signatur der Methode aufgeführt werden

```
String readLineFromFile(String filename) throws FileNotFoundException {
    // ...
}
```

- Compiler und Programmierer wissen, dass eine FileNotFoundException geworfen werden kann
- Programmierer kann angemessen darauf reagieren: Zum Beispiel Hinweis an Benutzer, dass die Datei nicht existiert

# Fangen einer Exception

Fangen einer Exception:

```
try { /* Code der Ausnahme produziert */ } catch (Throwable e) { /* Behandlung */ }
```

- Es *kann* jede Subklasse von Throwable gefangen werden
- Checked und unchecked Exceptions können gefangen werden
- Es können mehrere **catch** hintereinander stehen

# Fangen einer Exception

Fangen einer Exception:

```
try { /* Code der Ausnahme produziert */ } catch (Throwable e) { /* Behandlung */ }
```

- Es *kann* jede Subklasse von Throwable gefangen werden
- Checked und unchecked Exceptions können gefangen werden
- Es können mehrere **catch** hintereinander stehen

Beispiel:

```
try {
  readLineFromFile("invalid file name");
} catch (FileNotFoundException e) {
    System.out.println("File does not exist, please choose another one!");
} catch (AnotherException e) {
    // ...
}
```

# Exceptions behandeln

Ist es sinnvoll jede Exception zu fangen?

**Exceptions nur fangen, wenn man die Fehlersituation bereinigen oder melden kann.**

# Exceptions behandeln

Ist es sinnvoll jede Exception zu fangen?

> **Exceptions nur fangen, wenn man die Fehlersituation bereinigen oder melden kann.**

**Checked** Behandeln, Benutzer über Fehler informieren[2], und ggf. nochmals mit neuem Wert versuchen

**Unchecked** Möglichkeiten:

- Fehler im Source beheben
- Benutzer informieren **und** Programmierer informieren: Stacktrace, Logausgabe, Bugreport
- Ignorieren, d.h. kein try-catch

[2] Natürlich mit klarer Fehlerbeschreibung

# Exceptions behandeln

Ist es sinnvoll jede Exception zu fangen?

> **Exceptions nur fangen, wenn man die Fehlersituation bereinigen oder melden kann.**

**Checked** Behandeln, Benutzer über Fehler informieren[2], und ggf. nochmals mit neuem Wert versuchen

**Unchecked** Möglichkeiten:

- Fehler im Source beheben
- Benutzer informieren **und** Programmierer informieren: Stacktrace, Logausgabe, Bugreport
- Ignorieren, d.h. kein try-catch

**Niemals** Exceptions fangen, und "verschlucken":

```
try { callMethod(); } catch (Exception e) { }
```

[2] Natürlich mit klarer Fehlerbeschreibung

# Exceptions: Print, rethrow und wrap

Stacktrace ausgeben:

```
catch (Exception e) {
  e.printStackTrace(); // prints exception to System.err.
}
```

Aktueller Zustand liefert ggf. weitere Infos zum Fehler:

```
catch (Exception e) {
  if (...) {
    // No idea what happened
    throw e;
  } else {
    // We have more information on what went wrong
    throw new MyOwnException("Explanation", e);
  }
}
```

# Stacktraces bei Rethrow und Wrapping

**Rethrow** **throw** e;
Original Stacktrace bleibt erhalten

**Wrapper** **throw new** MyOwnException(e);
Erweiterter Stacktrace:

```
Exception in thread "main" MyOwnException: \
    java.lang.ArithmeticException: / by zero
        at Rethrow.wrapping(Rethrow.java:9)
        at Rethrow.main(Rethrow.java:14)
Caused by: java.lang.ArithmeticException: / by zero
        at Statistics.average(Statistics.java:20)
        at Rethrow.wrapping(Rethrow.java:7)
        ... 1 more
```

# Wichtige Exceptions

Checked:

- IOException (FileNotFoundException, EOFException, ServletException, . . . )
- ClassNotFoundException

Unchecked: Alle Subklassen von **RuntimeException**

- ArithmeticException
- NullPointerException
- IllegalArgumentException
- NoSuchElementException
- IllegalStateException
- IndexOutOfBoundsException
- ClassCastException
- UnsupportedOperationException

# Testen von Exceptions

```
public static double squareRoot(double x) {
    if (x < 0) {
        throw new IllegalArgumentException
            ("Roots from negative numbers not supported");
    }
}
```

# Testen von Exceptions

```
public static double squareRoot(double x) {
    if (x < 0) {
        throw new IllegalArgumentException
            ("Roots from negative numbers not supported");
    }
}
```

Klassisch:

```
@Test
public void SquareRootArgumentTest() {
  try {
    MyMath.squareRoot(-1);
    fail("No IllegalArgumentException thrown!");
  } catch (IllegalArgumentException e) {
    // ok
  }
}
```

# Testen von Exceptions (2)

Annotationsbasiert:

```
@Test(expected = IllegalArgumentException.class)
public voidSquareRootArgumentTest() {
  MyMath.squareRoot(−1);
}
```

# Testen von Exceptions (2)

Annotationsbasiert:

```
@Test(expected = IllegalArgumentException.class)
public voidSquareRootArgumentTest() {
  MyMath.squareRoot(−1);
}
```

Vorsicht:

```
@Test(expected = IllegalArgumentException.class)
public voidSquareRootArgumentTest() {
  // Was, wenn hier eine IllegalArgumentException geworfen wird?
  SomeMath aMathObject = new SomeMath(0.0000001);

  // Eigentliche Methode die getestet werden soll
  aMathObject.squareRoot(−1);
}
```

# Interfaces und Exceptions

- ActionListener implementiert, der auf Button click reagieren soll
- In `actionPerformed` with eine Methode aufgerufen die eine checked Exception wirft

```
public class MyWindow implements ActionListener {

  @Override
  public void actionPerformed(ActionEvent e) /* throws Exception */ {
      method(); // throws Exception
  }
```

# Interfaces und Exceptions

- ActionListener implementiert, der auf Button click reagieren soll
- In `actionPerformed` with eine Methode aufgerufen die eine checked Exception wirft

```
public class MyWindow implements ActionListener {

  @Override
  public void actionPerformed(ActionEvent e) {
      try {
          method();
      } catch (Exception e) {
        // Either handle or
        throw new RuntimeException(e);
      }
  }
```

# Interfaces und Exceptions

- ActionListener implementiert, der auf Button click reagieren soll
- In `actionPerformed` with eine Methode aufgerufen die eine checked Exception wirft

```
public class MyWindow implements ActionListener {

  @Override
  public void actionPerformed(ActionEvent e) {
      try {
          method();
      } catch (Exception e) {
        // Either handle or
        throw new RuntimeException(e);
      }
  }
```

- Behandeln (`try-catch`)
- In `RuntimeException` verpacken

# Zusammenfassung

- 3 Arten von Fehler
- Throwable, Error, Exception, RuntimeException
- Werfen und Behandeln von Fehlern
- Checked und Unchecked Exceptions
- Testen von Exceptions
- Interfaces und Exception

# Ein/Ausgabe

# Motivation

- Ein- und Ausgabe (input/output) wesentlicher Bestandteil von Programmen
- Bekanntes Beispiel:

```
System.out.println("Hello world");
```

  System.out ist eine Instanz eines PrintWriter.
- PrintWriter ist Teil einer Klassenhierarchie für Ein-/Ausgabe

# I/O in Java

- Abstraktion über Eingabe- und Ausgabemedium
- Javas Klassenhierarchie für IO ist im Package `java.io`
- Unterscheidung zwischen Ein-/Ausgabe von Binärdaten und Text
  Teilweise keine klare Trennung

# I/O in Java

- Abstraktion über Eingabe- und Ausgabemedium
- Javas Klassenhierarchie für IO ist im Package `java.io`
- Unterscheidung zwischen Ein-/Ausgabe von Binärdaten und Text
  Teilweise keine klare Trennung

Abstrakte Superklassen:

Binärdaten `java.io.InputStream` und `java.io.OutputStream`

Text `java.io.Reader` und `java.io.Writer`

# InputStream

| *InputStream* |
| --- |
| *read*() : int<br>close() : void<br>... |

read:

- `read` liest ein `Byte` (0-255) aus dem Stream
- Rückgabewerte -1 signalisiert Ende des Streams
- **throws** IOException

close:

- `close` schließt den Stream und gibt interne Resourcen frei.
- Muss nach Ende des Lesens aufgerufen werden.
- Stream kann nicht wieder geöffnet werden.

# Close und Exceptions

Problem:

- “close muss nach Ende des Lesens aufgerufen werden”.
- read() **throws** IOException und close() **throws** IOException

# Close und Exceptions

Problem:

- "close muss nach Ende des Lesens aufgerufen werden".
- read() **throws** IOException und close() **throws** IOException

Lösung: try-finally

```
InputStream in = null;
try {
    in = new FileInputStream("");
    in.read();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (in != null) {
        try {
            in.close();
        } catch (IOException e) { }
    }
}
```

# OutputStream

| *OutputStream* |
| --- |
| *write*(*b* : *int*) : void |
| close() : void |
| flush() : void |
| ... |

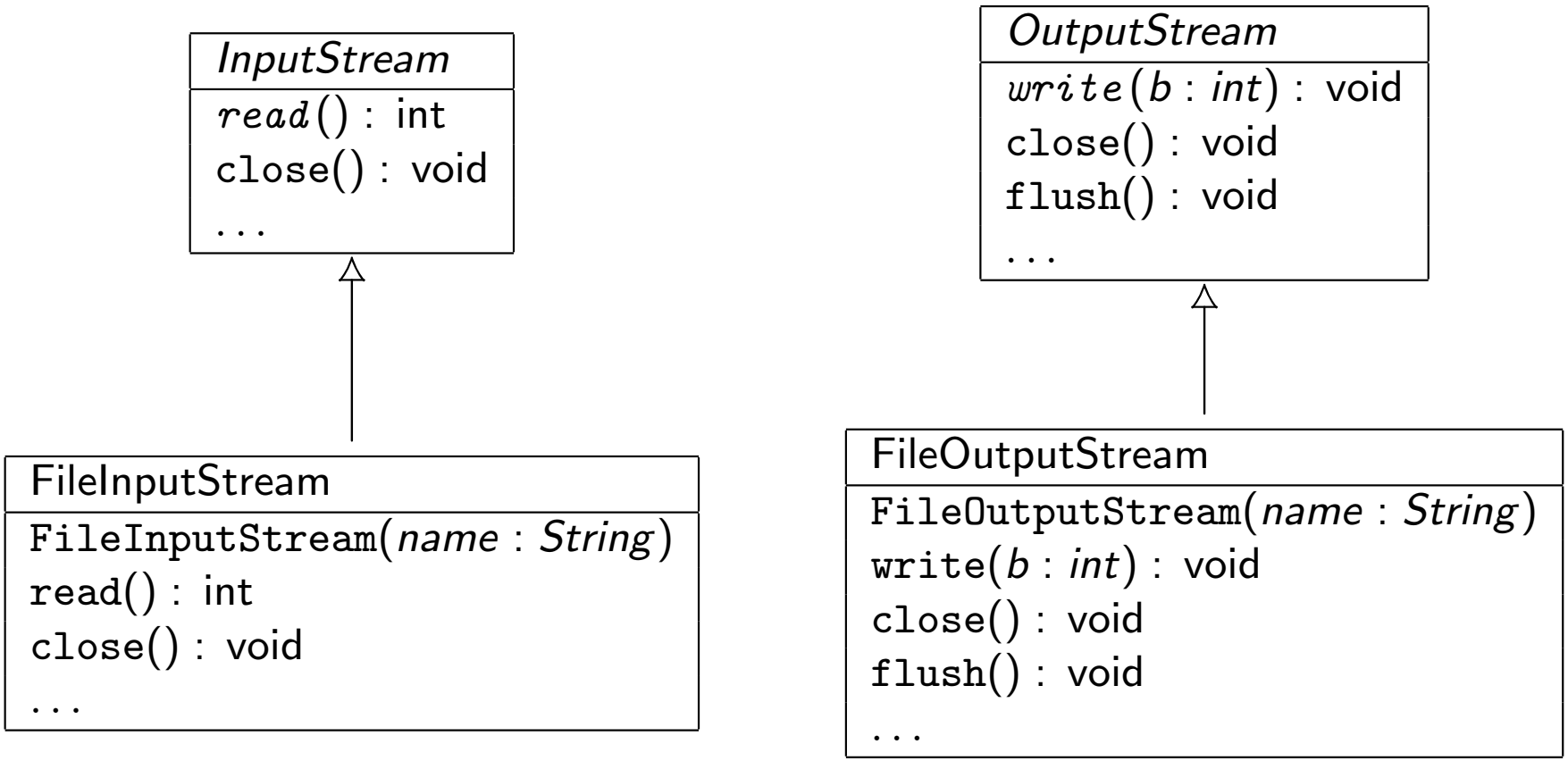
write:

- `write` schreibt ein `Byte` (0-255) in den Stream
- `write(01); write(0xFF01); write(256);` identisch
- **throws** IOException

flush:

- Aus Effizienzgründen werden die Daten oft gepuffert
- `flush` schreibt alle gepufferten Daten in den Stream

# Konkrete Klassen zur Ein/Ausgabe von Dateien



# Beispiel: Kopieren einer Datei

```
public static void copyAll(InputStream in, OutputStream out)
    throws IOException {
  int b;
  while ((b = in.read()) != -1) {
    out.write(b);
  }
}

public static void main(String[] args) throws IOException {
  InputStream from = new FileInputStream("/tmp/test.bin");
  OutputStream to = new FileOutputStream("/tmp/test_copy.bin");
  copyAll(from, to);
  from.close();
  to.close();
}
```

# Beispiel: Kopieren einer Datei

```
public static void copyAll(InputStream in, OutputStream out)
      throws IOException {
  int b;
  while ((b = in.read()) != -1) {
    out.write(b);
  }
}

public static void main(String[] args) throws IOException {
  InputStream from = new FileInputStream("/tmp/test.bin");
  OutputStream to = new FileOutputStream("/tmp/test_copy.bin");
  copyAll(from, to);
  from.close();
  to.close();
}
```

- **Aber:** Nicht effizient, da immer nur 1 Byte gelesen und geschrieben wird!
- **Lösung:** Benutzung von puffernden Filterklassen

# Motivation

## Effizientes Kopieren mit puffernden Filterklassen

**Idee**: Einen Puffer, der sich wie ein Stream verhält, dem eigentlichen Stream vorschalten.

# Motivation

## Effizientes Kopieren mit puffernden Filterklassen

**Idee**: Einen Puffer, der sich wie ein Stream verhält, dem eigentlichen Stream vorschalten.

Ersetze

```
InputStream from = new FileInputStream("test.bin");
OutputStream to = new FileOutputStream("test2.bin");
```

durch

```
InputStream from = new BufferedInputStream(new FileInputStream("test.bin"));
OutputStream to = new BufferedOutputStream(new FileOutputStream("test2.bin"));
```

# Filterklassen

Ein Eingabefilter arbeitet auf einem InputStream und verhält sich wie ein InputStream.

| BufferedInputStream |
| --- |
| `BufferedInputStream`(*in* : *InputStream*) |
| `read`() : int |
| `close`() : void |
| ... |

# Intermezzo: Entwurfsmuster

Entwurfsmuster (Design patterns):

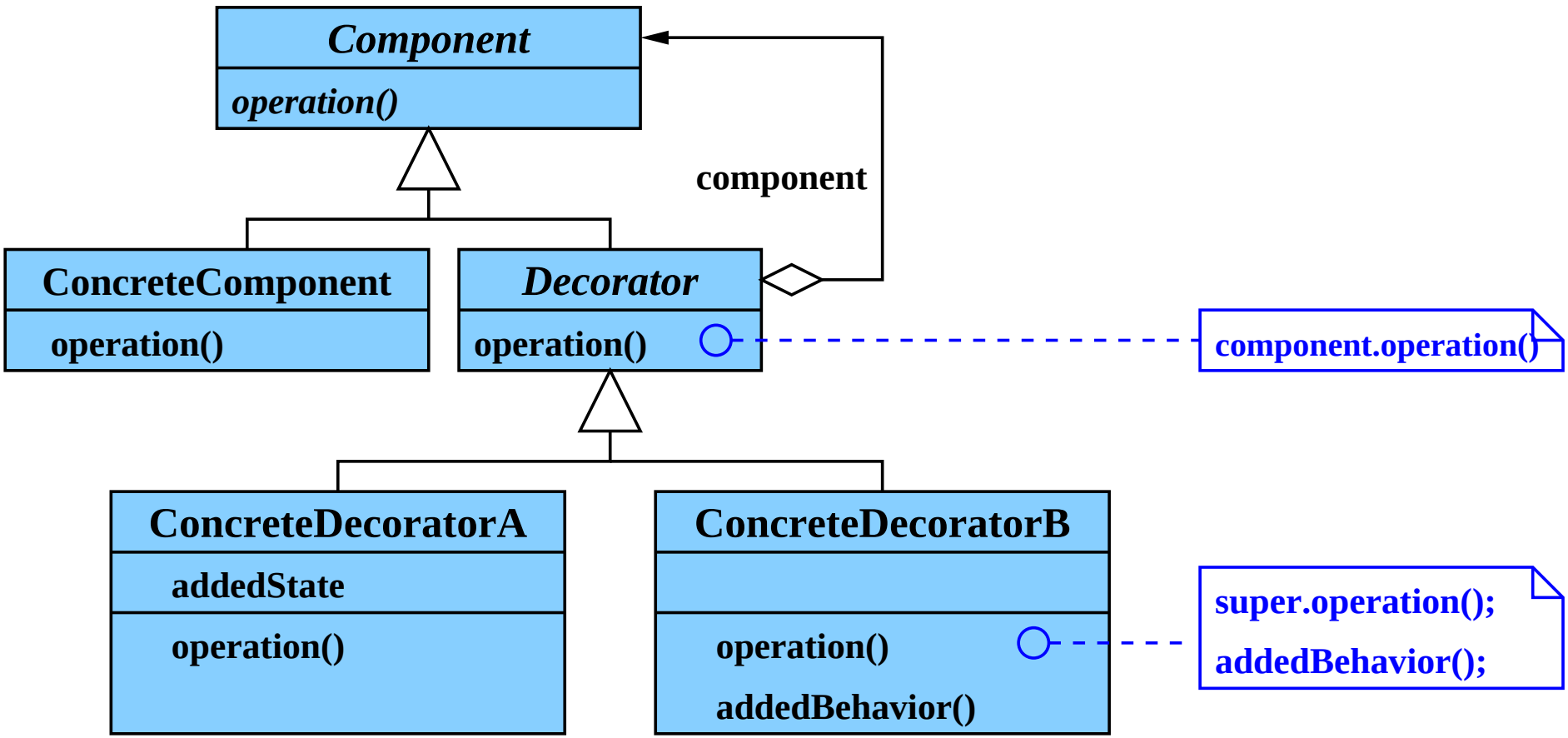
- Standardisierte Vorlagen um wiederkehrende Problem zu lösen
- Gamma, Helm, Johnson, Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995
- Hier nur Ausschnitt: Vorlesung Softwaretechnik

# Decorator

Decorator

- Dynamische Erweiterung einer Klasse um Funktionalität
  Hier: FileInputStream durch Puffer erweitert
- Methodenaufrufe:
  - Veränderung des Verhaltens
  - Weiterleitung (Delegation)
- Unsichtbar: Aufrufer merkt nicht, dass ein Decorator vorgeschalten ist

# Klassendiagramm Decorator



# Filterklassen in Java

Basisklasse

- FilterInputStream ist Superklasse aller Eingabefilter
- Standardverhalten: Weiterleitung an dekorierte Komponente

Eine Auswahl:

- BufferedInputStream
- CipherInputStream
- GZIPInputStream
- ZipInputStream

Hinweis: Es existiert auch eine Klassenhierarchie für FilterOutputStream

# Beispielfilter

Alle gelesenen Bytes um 1 erhöhen

```
public class IncInputStream extends FilterInputStream {

  protected IncInputStream(InputStream in) {
    super(in);
  }

  @Override
  public int read() throws IOException {
    int value = in.read();

    if (value == -1) {
      return value;
    } else {
      return (value + 1) % 0x100;
    }
  }
}
```

# Character streams

| *Reader* |
| --- |
| *read*() : int |
| *close*() : void |
| read(*cbuf* : *char*[]) : int |
| ... |

| *Writer* |
| --- |
| *write*(*c* : *int*) : void |
| *close*() : void |
| write(*cbuf* : *char*[]) : void |
| write(*str* : *String*) : void |
| append(*c* : *char*) : Writer |
| ... |

# Klassenhierarchie Character streams (Auswahl)

Reader

- FileReader
- StringReader

Writer

- FileWriter
- StringWriter
- PrintWriter

# Filterklassen

- FilterReader als Basisklasse (vgl. FilterInputStream)
- Aber: BufferedReader nicht als FilteredReader implementiert!
  BufferedReader hat eine Methode `String readLine()`

Hinweis: Gilt auch für Writer

# Datei zeilenweise einlesen

```
BufferedReader r = new BufferedReader(new FileReader("/tmp/test.txt"));

String line;
while ((line = r.readLine()) != null) {
  System.out.println(line);
}
r.close();
```

# Scanner

- Interface des `Reader` ist ziemlich primitiv
- Klasse `java.util.Scanner` zum einlesen von primitiven Datentypen
- Scanner ist ein Iterator (kein Decorator)
- Trennzeichen: Whitespace (Leerzeichen, Tab, Newline)

# Scanner

- Interface des `Reader` ist ziemlich primitiv
- Klasse `java.util.Scanner` zum einlesen von primitiven Datentypen
- Scanner ist ein Iterator (kein Decorator)
- Trennzeichen: Whitespace (Leerzeichen, Tab, Newline)

```
Reader r = ...;
Scanner scanner = new Scanner(r);

while (scanner.hasNext()) {
    System.out.println(scanner.next());
}
scanner.close();
r.close();
```

# Scanner

- Interface des `Reader` ist ziemlich primitiv
- Klasse `java.util.Scanner` zum einlesen von primitiven Datentypen
- Scanner ist ein Iterator (kein Decorator)
- Trennzeichen: Whitespace (Leerzeichen, Tab, Newline)

```
Reader r = ...;
Scanner scanner = new Scanner(r);

while (scanner.hasNext()) {
    System.out.println(scanner.next());
}
scanner.close();
r.close();
```

Methoden `boolean hasDatentyp()` und `Datentype nextDatentyp()`

# PrintWriter

- Formatierte Ausgaben von primitiven Datentypen
- Für Objekte: ‘‘`null`’’ oder `obj.toString()`
- Methoden werfen keine `IOException`
- Ruft nicht automatisch flush() auf

  **new** PrintWriter(writer, true);

# Encoding

- FileReader benutzt das Standard Encoding des OS (`System.out.println(Charset.defaultCharset());`)
- Encoding: Abbildung eines Zeichen in eine Folge von Bytes

Bekannte Encodings:

- US-ASCII, ISO-8859-1 (Latin-1)
- Unicode: UTF-8, UTF-16

# FileReader mit Encoding

Datei mit anderem Encoding einlesen

- Brücke zwischen InputStream und FileReader: InputStreamReader bzw. OutputStreamWriter
- Encoding als String (siehe Javadoc) oder Charset

```
BufferedReader r = new BufferedReader(new InputStreamReader(
        new FileInputStream("/tmp/test.txt"), "ISO-8859-1"));

String line;
while ((line = r.readLine()) != null) {
  System.out.println(line);
}
r.close();
```

# Testen von IO

- JUnit-Tests mit externen Resourcen ist schwierig
- Simulation über String

## String als Pseudoinputstream

```
String myTestString = "...";
InputStream stream = new ByteArrayInputStream(myTestString.getBytes("UTF-8"));
```

## String als Reader

```
String myTestString = "...";
Reader r = new StringReader(myTestString);
```

# Zusammenfassung

- Unterscheidung zwischen Binärdaten und Textdaten
  - InputStream, OutputStream
  - Reader, Writer
- Filterklassen als Decorator
- Encoding bei Textdaten
- Testen von IO

# Serialisierung nach XML

# Motivation

Serialisierung

- Objekte existieren nur während das Programm läuft
- Erhalten von Objekten über verschiedene Programmabläufe hinweg (Persistenz)
- Serialisierung: Objekte, speziell deren Feldinhalte, in Dateien speichern
- Deserialisierung: Wiederherstellen von Objekten aus Dateien
- Kleiner Ausschnitt: Serialisierung nach XML

# XML eine minimale Einführung

- Linearisierte Darstellung einer Baumstruktur
- XML-Dokument ist lesbar (enthält keine Binärdaten)

```
<list>
  <book title="Hamlet">
    <author>
      <firstname>William</firstname>
      <lastname>Shakespeare</lastname>
    </author>
  </book>
  <book title="The Tempest">
    <author>
      <firstname>William</firstname>
      <lastname>Shakespeare</lastname>
    </author>
  </book>
</list>
```

# XStream

Es existieren viele Libraries zur Serialisierung nach XML

- Java API for XML Binding (JAXB)
  - Offizielle Lösung (`http://jaxb.java.net`)
  - Erfordert aber Änderungen an Klassen
- XStream
  - Entwickelt von Codehaus
  - liberale BSD-Lizenz
  - Klassen ohne Anpassung serialisierbar

Hier: XStream

# Beispiel

Package: `example.serialization`

| Book |
| --- |
| `name` : String<br>`author` : Author |
| `Book(name` : String, `author` : Author)<br>... |

| Author |
| --- |
| `firstname` : String<br>`lastname` : String |
| `Author(firstname` : String, `lastname` : String)<br>... |

# Beispiel - Serialisierung mit XStream

Vorbereiten:

```
Author author = new Author("William", "Shakespeare");
Book hamlet = new Book("Hamlet", author);
Book tempest = new Book("The Tempest", author);

List<Book> books = new ArrayList<Book>();
books.add(hamlet);
books.add(tempest);
```

# Beispiel - Serialisierung mit XStream

Vorbereiten:

```
Author author = new Author("William", "Shakespeare");
Book hamlet = new Book("Hamlet", author);
Book tempest = new Book("The Tempest", author);

List<Book> books = new ArrayList<Book>();
books.add(hamlet);
books.add(tempest);
```

Serialisieren:

```
XStream xstream = new XStream(new DomDriver());

String xml = xstream.toXML(books);
```

# Beispiel - Serialisierung mit XStream

Vorbereiten:

```
Author author = new Author("William", "Shakespeare");
Book hamlet = new Book("Hamlet", author);
Book tempest = new Book("The Tempest", author);

List<Book> books = new ArrayList<Book>();
books.add(hamlet);
books.add(tempest);
```

Serialisieren:

```
XStream xstream = new XStream(new DomDriver());

String xml = xstream.toXML(books);
```

```
Writer w = ...;
xstream.toXML(books, w);

OutputStream out = ...;
xstream.toXML(books, out);
```

# Ausgabe

```
<list>
  <example.serialization.Book>
    <name>Hamlet</name>
    <author>
      <firstname>William</firstname>
      <lastname>Shakespare</lastname>
    </author>
  </example.serialization.Book>
  <example.serialization.Book>
    <name>Temptest</name>
    <author reference="../../example.serialization.Book/author" />
  </example.serialization.Book>
</list>
```

# Aliaseinführung

- XStream nutzt vollqualifizierten Klassennamen
- Aliase zur Abkürzung
  - xstream.alias("book", Book.**class**);
    Ersetze Elementname der Book Klasse durch `book`

# Aliaseinführung

- XStream nutzt vollqualifizierten Klassennamen
- Aliase zur Abkürzung
  - xstream.alias("book", Book.**class**);
    Ersetze Elementname der Book Klasse durch `book`
  - xstream.aliasField("title", Book.**class**, "name");
    Ersezte Feldname `name` durch `title`

# Aliaseinführung

- XStream nutzt vollqualifizierten Klassennamen
- Aliase zur Abkürzung
  - xstream.alias("book", Book.**class**);
    Ersetze Elementname der Book Klasse durch `book`
  - xstream.aliasField("title", Book.**class**, "name");
    Ersezte Feldname `name` durch `title`
  - xstream.aliasAttribute(Book.**class**, "name", "title");
    Schreibe das Feld `name` der Klasse `Book` als Attribut mit Namen `title`.

# Zyklische Objektstrukturen

| Author |
| --- |
| `firstname` : String<br>`lastname` : String<br>`books` : List⟨Book⟩ |
| `Author(firstname` : String, `lastname` : String)<br>... |

- XML: Linearisierte Darstellung einer Baumstruktur
- Zyklen in **Objektstrukturen** nicht als Baum darstellbar!

# Zyklische Objektstrukturen in XStream

- XStream erkennt zyklische Objektstrukturen
- Verwendet Referenzen in XML:

```
<list>
  <book title="Hamlet">
    <author>
      <firstname>William</firstname>
      <lastname>Shakespare</lastname>
      <books>
        <book reference="../../.."/>
        <book title="Temptest">
          <author reference="../../.."/>
        </book>
      </books>
    </author>
  </book>
  <book reference="../book/author/books/book[2]"/>
</list>
```

# Zyklische Objektstrukturen in XStream

Über xstream.setMode kontrollierbar

- Relative XPATH-Referenzen (Standard)
  XStream.XPATH_RELATIVE_REFERENCES

# Zyklische Objektstrukturen in XStream

Über xstream.setMode kontrollierbar

- Relative XPATH-Referenzen (Standard)
  XStream.XPATH_RELATIVE_REFERENCES
- Keine Referenzen
  XStream.NO_REFERENCES
  - Jedes Objekt wird bei jedem vorkommen serialisiert
  - Objekte tauchen u.U. mehrfach im XML auf
  - Exception wenn Zyklus vorhanden

# Zyklische Objektstrukturen in XStream

Über xstream.setMode kontrollierbar

- Relative XPATH-Referenzen (Standard)
  XStream.XPATH_RELATIVE_REFERENCES
- Keine Referenzen
  XStream.NO_REFERENCES
  - Jedes Objekt wird bei jedem vorkommen serialisiert
  - Objekte tauchen u.U. mehrfach im XML auf
  - Exception wenn Zyklus vorhanden
- XML-Element Nummerierung
  XStream.ID_REFERENCES
  - Jedes Element erhält eine eindeutige ID
  - Referenzierung über diese ID:
    <book reference="2" />

# Deserialisierung

- Generierung von Objektstrukturen aus XML
- Klassen müssen vorhanden sein
- Gleiche Aliasangaben

```
List<Book> books = (List<Book>) xstream.fromXML(xml);

InputStream in = ...;
List<Book> books2 = (List<Book>) xstream.fromXML(in);

Reader r = ...;
List<Book> books3 = (List<Book>) xstream.fromXML(r);
```

# Zusammenfassung

- Serialisierung: Speichern von Objekten zur späteren Deserialisierung
- Serialisierung nach/von XML am Beispiel von XStream
- Vorsicht bei zyklischen Objektstrukturen