Exceptions und I/O-Streams in Java

PROG 2: Einführung in die Programmierung für Wirtschaftsinformatiker

Steffen Helke

Technische Universität Berlin
Fachgebiet Softwaretechnik

29. April 2013

Übersicht

- Wiederholung: Ereignisbehandlung in einer GUI
- Fehlerbehandlung mit Exceptions
- Ein- und Ausgabestreams

Teil I der Vorlesung PROG 2

Entwicklung grafischer Schnittstellen

Graphical **U**ser **I**nterface

– Ereignisbehandlung in einer GUI –

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

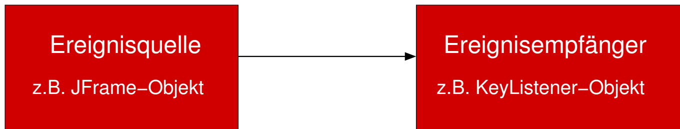
Ereignisbehandlung (Event handling)

Intention

- Festlegung, was passieren soll, wenn GUI-Eingaben erfolgen

Konzepte zur Umsetzung

- Erzeugung von Ereignissen (*Events*, z.B. ESC-Taste Drücken)
- Belauschen der Ereignisse mit speziellen Objekten (*Listener*)
- Definition von *Eventcode* (Ereignisbehandlung)



Entwurfsmuster zur Ereignisbehandlung

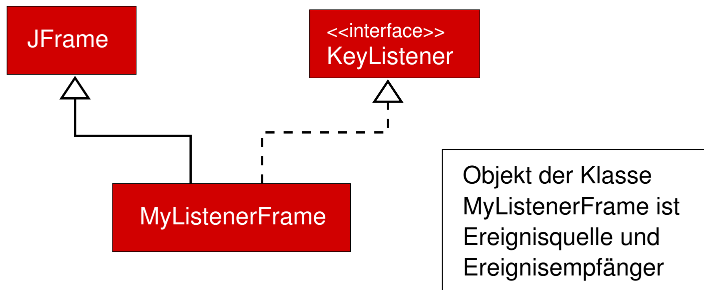
Varianten

1. Fensterklasse implementiert erforderliche Interfaces für EventListener und registriert sich selbst bei Ereignisquellen
2. Definition von lokalen oder anonymen Klassen in der Fensterklasse, um EventListener zu implementieren
3. Trennung von GUI-Code und Ereignisbehandlung in komplett separaten Klassen
4. Überlagerung spezieller Methoden der Komponentenklasse, die für Empfangen/Verteilen von Nachrichten erforderlich sind

Trennung von GUI-Code und Ereignisbehandlung ist im Sinne des MVC-Patterns für größere Programme unbedingt zu empfehlen!

Variante 1: Entwurf zur Ereignisbehandlung

Fenster-Klasse implementiert EventListener-Interface



Variante 1: Entwurf zur Ereignisbehandlung

Fenster-Klasse implementiert EventListener-Interface

⇒ einfacher Zugriff auf alle Methoden in einer Klasse

⇒ Nachteil: unübersichtlich und viele leere Methoden

```
class MyFrame extends JFrame implements KeyListener {
  public static void main(String[] args) { ... }
  public MyFrame() {
          ...;  addKeyListener(this);
          }
  public void keyPressed(KeyEvent event) {
          if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
              setVisible(false);
              dispose();
              System.exit(0);
            }
          }
  public void keyReleased(KeyEvent event) {  }
  public void keyTyped(KeyEvent event) {  }
}
```

Adapter-Klassen zur Ereignisbehandlung

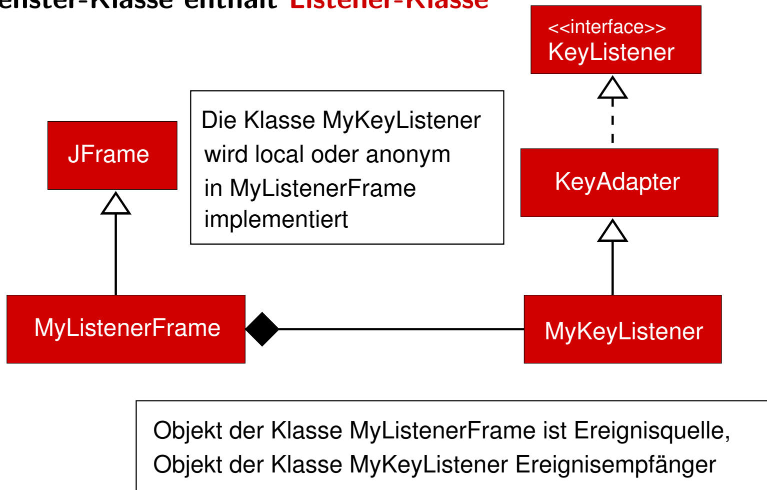
Intention

- Häufig sind nicht alle Ereignisbehandlungen aus einem EventListener-Interface für die Anwendung interessant
- *Adapterklassen* implementieren für zugehörige Interfaces leere Methoden
- Eigene *Listener*-Klassen können von Adapter-Klassen erben und nur für die Anwendung interessante Methoden definieren

```
public abstract class WindowAdapter
    implements WindowListener, ... {
      public void windowOpened (WindowEvent e) {  };
      public void windowClosing (WindowEvent e) {  };
      public void windowClosed (WindowEvent e) {  };
      public void windowActivated (WindowEvent e) { };
      ...
  }
```

Variante 2: Entwurf zur Ereignisbehandlung

Fenster-Klasse enthält Listener-Klasse



Variante 2: Entwurf zur Ereignisbehandlung

Fenster-Klasse enthält lokale Klassen

- ➜ Kapselung der Event-Funktionalität in einer lokalen Klasse, die von *Adapterklasse* erbt
- ➜ Vorteil: nur benötigte Methoden zu implementieren, aber Ereignisquelle/Ereignisbearbeitung in der selben Klasse

```
class MyFrame extends JFrame {
  public static void main(String[] args) { ... }
  public MyFrame() {
        ...;
        addKeyListener(new MyKeyListener());}
  private class MyKeyListener extends KeyAdapter {
        public void keyPressed(KeyEvent event) {
          if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
            setVisible(false);dispose();System.exit(0);
  }}}}
```

Variante 2: Entwurf zur Ereignisbehandlung

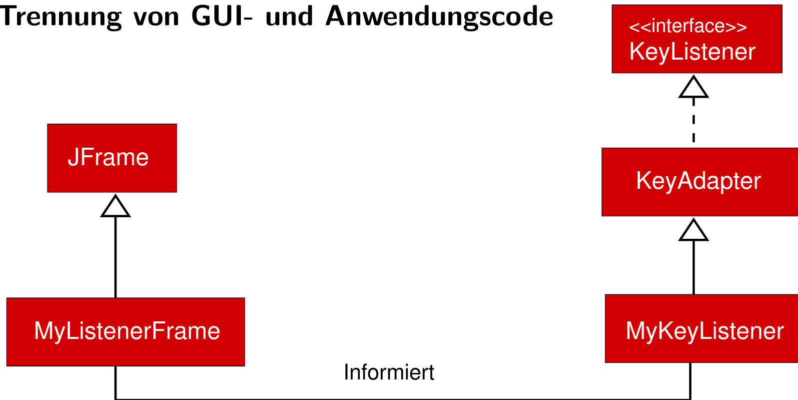
Fenster-Klasse enthält anonyme Klasse

- ➜ Kapselung der Event-Funktionalität in einer anonymen Klasse, die von *Adapterklasse* erbt
- ➜ Vorteil: weniger Code, aber noch weniger Kapselung, keine Trennung von GUI und Anwendungslogik

```
class MyFrame extends JFrame {
  public static void main(String[] args) { ... }
  public MyFrame() { ...;
    addKeyListener(
      new KeyAdapter() {
          public void keyPressed(KeyEvent event) {
            if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
              setVisible(false);dispose();System.exit(0);}}
          });
  }
```

Variante 3: Entwurf zur Ereignisbehandlung

Trennung von GUI- und Anwendungscode



Klasse MyKeyListener ist nicht lokal, sondern öffentlich definiert

Variante 3: Entwurf zur Ereignisbehandlung

Trennung von GUI- und Anwendungscode

- Implementierung der Ereignisbehandlung in zwei separaten Klassen ⇒ bessere Modularisierung des Codes
- Verbindung durch Aufruf der Methode *addKeyListener*

```
class TestListenerFrame {
  public static void main(String[] args) {
    KeyListener listener = new MyKeyListener();
    JFrame window = new MyListenerFrame(listener); }}

class MyListenerFrame extends JFrame {
  public MyListenerFrame(KeyListener listener) {
    super("Listener-Variante-3");
    addKeyListener(listener);  ...  }

class MyKeyListener extends KeyAdapter {
  public void keyPressed (KeyEvent event) {
     if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
          JFrame frame = (JFrame) event.getSource();
          frame.setVisible(false); frame.dispose(); ... }}}
```

Variante 4: Entwurf zur Ereignisbehandlung

Überlagerung Event-Handler in den Komponenten

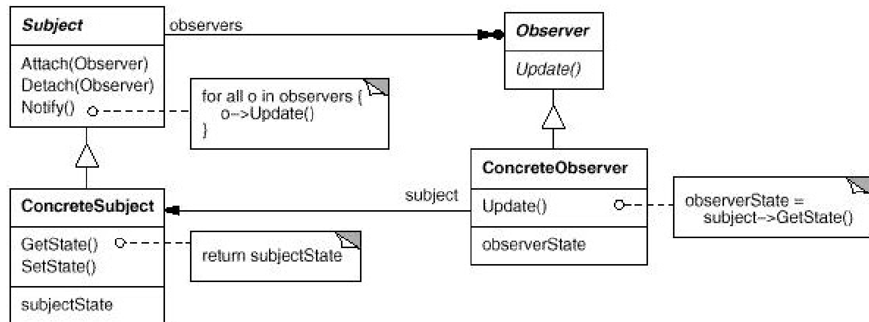
- Ereignisquellen haben Methoden zur Nachrichtenverteilung
- Weiterreichen von Nachrichten mit der Methode *processEvent*
- Verteilung durch *processEvent* anhand des Nachrichtentyps an Spezialmethoden (z.B. *processKeyEvent*), die bei Variante 4 überschrieben werden ➜ Listener-Pattern wird umgangen!

```
class MyListenerFrame extends JFrame {
  public MyListenerFrame() {
    super("Listener-Variante-4");
    setSize(300, 300); ... }

  public void processKeyEvent(KeyEvent event)  {
     if (event.getID() == KeyEvent.KEY_PRESSED) {
       if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
             setVisible(false); ... }}
     super.processKeyEvent(event);}
```

Listener-Pattern ist Spezialform des Observer-Pattern

➜ Java bietet mit dem Interface *Observer* und der Klasse *Observable* auch Möglichkeiten zur direkten Umsetzung eines Observer-Pattern



Teil II der Vorlesung PROG 2

Ausnahmebehandlung (Exception Handling)

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Ausnahmen (Exceptions)

Arten von Laufzeitfehlern

1. logische Programmierfehler (z.B. Division durch 0)
2. fehlerhafte Bedienung eines korrekten Programms (z.B. kein Leserecht für ausgewählte Datei)
3. Probleme im Java-Laufzeitsystem

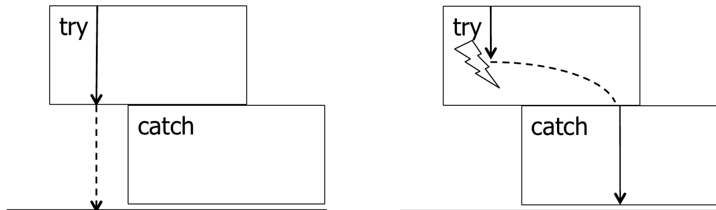
Varianten der Fehlerbehandlung

- *Maskieren*: Verbergen/Abschwächen eines Fehlers (z.B. durch Wiederholen gescheiterter Aktionen)
- *Tolerieren*: Fehleranzeige und Benutzerreaktion abwarten (z.B. Web-Site nicht erreichbar)
- *Wiederherstellen*: Rekonstruktion konsistenter Systemzustände

Java-Prinzip zur Ausnahmebehandlung

Try-Catch-Konzept

- Unterteilung des Codes in *try*- und *catch*-Blöcke
- (potenziell fehlerhafter) Programmcode in *try*-Blöcken
- Fehlerbehandlung in *catch*-Blöcken
- nur im Fehlerfall wird *catch*-Block abgefragt



Java-Schlüsselwörter zur Fehlerbehandlung

try

- enthält potenziell fehlerhaften Programmcode

catch

- behandelt aufgetretene Exceptions

finally

- definiert Aufräumarbeiten nach Verlassen von *try*-*catch*-Block

throws

- deklariert unbehandelte Exceptions in der Methodensignatur

throw

- löst eine Exception aus

Abstrakte Formulierung von Try und Catch

```
try {
      Programmfragment mit potenziellen Fehlern
    }
```

Abstrakte Formulierung von Try und Catch

```
try {
        Programmfragment mit potenziellen Fehlern
    }

catch (Exception-Art 1 e1) {
        Fehlerbehandlung 1
    }
...
catch (Exception-Art n en) {
        Fehlerbehandlung n
    }
```

Abstrakte Formulierung von Try und Catch

```
try {
      Programmfragment mit potenziellen Fehlern
    }

catch (Exception-Art 1 e1) {
      Fehlerbehandlung 1
    }
...
catch (Exception-Art n en) {
      Fehlerbehandlung n
    }

finally {
      Abschlussaktivitaet (falls noetig)
    }
```

Hinweis: Nur die erste passende catch-Anweisung wird ausgeführt, alle nachfolgenden (bis auf *finally*) werden ignoriert!

Finally-Blocks

Intention

- Aufräumarbeiten nach Abschluss, um nicht mehr benötigte Ressourcen freizugeben
- gemeinsamen Code aus *catch*-Anweisungen und *try*-Anweisung zusammenfassen (redundanten Code vermeiden)

Wann wird ein finally-Block ausgeführt?

1. Exception ist nicht aufgetreten und *try*-Block wurde erfolgreich ausgeführt
2. Exception ist aufgetreten und passende *catch*-Anweisung wurde ausgeführt
3. Exception ist aufgetreten, aber keine passende *catch*-Anweisung gefunden

Deklaration von möglichen Exceptions

Verpflichtung

- Deklaration möglicher Exceptions ist verpflichtend, wenn sie nicht im *try-catch*-Block einer Methode behandelt werden

⇒ Erweiterung der Methodensignatur mit *throws*-Statement

```
return_type Methodenname (Parameter)
            throws Exception1,...,ExceptionN
```

Ausgenommen sind ...

- Exceptions, die überall im Programm auftreten können, z.B. Division durch 0 oder nicht definierte Array-Zugriffe
- verpflichtende Behandlung würde Code unleserlich machen

⇒ betrifft alle Unchecked Exceptions in Java, d.h. die Subklassen von *RuntimeException* und *Error*

Auswertung einfacher try-catch-Blöcke

Regeln

1. erste passende *catch*-Anweisung wird genommen, alle anderen *catch*-Anweisungen ignoriert
2. keine passende *catch*-Anweisung ⇒ Abbruch des Programms

```
class ButtonsListener implements ActionListener {
 private String laf;
 public void actionPerformed(ActionEvent e) {
   if(e.getSource() == metalButton) {
       laf = "javax.swing.plaf.metal.MetalLookAndFeel"; }
   ...
   try { UIManager.setLookAndFeel(laf); }
   catch (UnsupportedLookAndFeelException ue) {
       System.err.println(ue.toString()); }
   catch (ClassNotFoundException ce) {
       System.err.println(ce.toString()); } ...
```

Auswertung geschachtelter try-catch-Blöcke

Aufbau

- Methode mit *try-catch*-Block ruft weitere Methoden mit eigenen *try-catch*-Blöcken auf
- Stack-basierte Implementierung, zuletzt geöffneter *try-catch*-Block bildet oberstes Element auf dem Stack
- Verlassen eines *try-catch*-Blockes ⇒ Entfernen des *try-catch*-Blocks vom Stack

Regeln

1. Exceptionbehandlung: Suche nach passender *catch*-Anweisung im gesamten Stack mit oberstem Element beginnend
2. kein passender *catch*-Block ⇒ Abbruch des Programms

Fehlerverfolgung in try-catch-Blöcken

Problem

- Fehlerverfolgung und Debugging in verschachtelten *try-catch*-Blöcken schwierig

Hilfsmittel zur Ausgabe

- Fehlerbeschreibung einer Exception

```
String getMessage()
```

- Informationen zur Stack-Darstellung

```
StackTraceElement[] getStackTrace()
```

- Ausgabe von Exception und Trace

```
void printStackTrace()
```

- Zeile, Methoden- oder Klassenname der Exception

```
getLineNumber(), getMethodName(), getClassName()
```

Wie benutze ich Exceptions richtig?

1. Einsatz von möglichst spezifischen Exceptions, keine sinnlosen Fehlerbehandlungen, wie z.B.

```
catch(Throwable e) {System.err.println("Exception");}
```

2. leere *catch*-Blöcke vermeiden
3. *RuntimeExceptions* nicht deklarieren oder abfangen, sondern durch geeigneten Code das Auftreten verhindern
4. bereits definierte Fehlerklassen nutzen und nur sparsam neue Fehlerklassen einführen
5. Dokumentation unter Verwendung von @throws aus Javadoc

```
/**
* @throws NewException
*      Erklaerender Text
**/
public void method (...) throws NewException { ... }
```

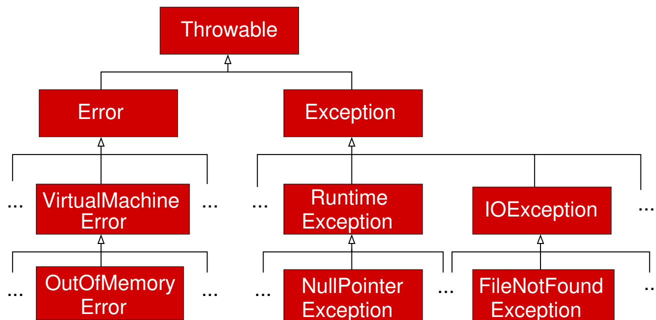
RuntimeException vs. Error

RuntimeException

- Programmierfehler, die vermeidbar sind (z.B. Division durch 0)

Error

- Probleme, die eigentlich nicht auftreten sollten (z.B. Fehler in der JVM)



Teil II der Vorlesung PROG 2

Ein- und Ausgabe mit Dateien

Allgemeine Einführung: Dateien und Dateisysteme

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Dateien und Dateisysteme

Schlüsselwörter

- Datei (*file*): Sammlung von Daten auf stabilem Speicher
- Verzeichnis (*directory, folder*): Sammlung von Dateien, in geeigneten Strukturen organisiert
- Partition (*partition*): Physische oder logische Aufteilung der Verzeichnisstrukturen

Definition Dateisystem

- Abstraktionsmechanismus, um Daten auf Speichermedium zugänglich zu machen

Eigenschaften von Dateisystemen

Merkmale eines Dateisystems

- Persistente Datenspeicherung
- Abstraktion von Details für Datenablage
- gleichzeitiger Zugriff durch mehrere Prozesse möglich
- Schutzmechanismen für Datenzugriffe

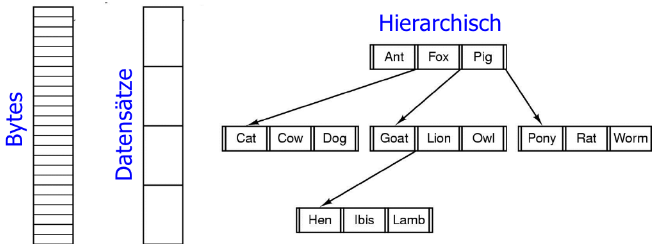
Dateiarten

1. *Textdatei*: zeilenweises Einlesen der Daten
2. *Binärdatei*: Bytesequenz in Speicher laden und Ausführen

Organisationsformen von Dateien

Strukturierungsmöglichkeiten

- Lineare Folge von Bits, Bytes: Dateistruktur bestimmt Anwendung und wird beim Einlesen interpretiert
- Sequenz von Datensätzen: Einlesen/Überschreiben zusammengesetzter Strukturen fester Größe
- Hierarchisch: Identifikation der Datensätze über Schlüssel



Dateiattribute (Auswahl)

1. *symbolischer Bezeichner*: vom Benutzer vergebener Name, Hinweis über Dateiinhalt, assoziierte Anwendung
2. *eindeutiger Bezeichner*: meist numerisch, Identifikation im Dateisystem
3. *Dateityp*: reguläre Datei (alphanumerisch oder binär), Verzeichnis, ...
4. *Position*: Zeiger auf den Speicherplatz der Datei
5. *Größe*: aktuelle Dateigröße in Bytes, Wörtern oder Blöcken
6. *Rechte*: Wer darf die Datei lesen, schreiben oder ausführen?
7. *Zeit & Datum*: Informationen über Erstellung, letzten Zugriff oder letzte Modifikation

Teil II der Vorlesung PROG 2

Ein- und Ausgabe mit Dateien

Dateiverarbeitung in Java

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Dateiverarbeitung in Java

Verarbeitungslevel

- Java setzt *keine Struktur* voraus
- Dateien werden als Folgen von Zeichen interpretiert
- Verarbeitung von Zeichenfolgen (*Streams*)

Vorgehen

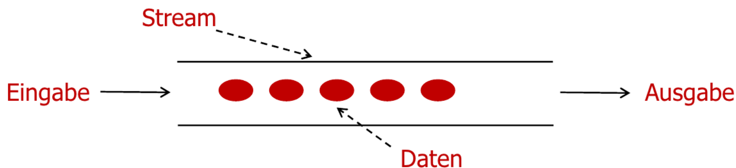
1. Öffnen der Datei

   ⇒ Verbindungsaufbau zwischen Datei und Programm
2. Bearbeiten der geöffneten Datei

   ⇒ Streams werden gelesen oder geschrieben
3. Schließen der Datei

   ⇒ Herstellung eines definierten Zustands

Streams in Java



Varianten

1. Byte-basierte Streams
   - Daten werden als Bytes gelesen/geschrieben
   - geeignet *für binäre Dateien*
2. Charakter-basierte Streams
   - Daten werden als alpahnumerische Zeichen (16 Bit Unicode-Zeichen) gelesen/geschrieben
   - geeignet *für textuelle Dateien*

Erzeugen einer Datei

Allgemeines

- Objekt der Klasse *File* repräsentiert eine Datei
- mögliche Konstruktoren

```
// Dateiname incl. Pfad
File (String name)
// Pfad und Dateiname getrennt
File (String path, String name)
// Pfad vom File dir
File (File dir, String name)
```

Erzeugen im Dateisystem

- Anlegen einer Datei, falls es diese noch nicht gibt
- Prüfen und Anlegen bilden atomare Operation
- Rückgabewert gibt an, ob Anlegen erfolgreich war

```
public boolean createNewFile() throws IOException
```

Lesen und Schreiben von Dateien

Herausforderung

- Auswahl der richtigen Werkzeuge
- *mehr als 20 Dateizugriffsklassen* verfügbar

Unterscheidung

1. Sequentieller Zugriff (*stream access*): Elemente als Zeichenfolge, Bearbeitung erfolgt streng nacheinander
2. Wahlfreier Zugriff (*random access*): Dynamische Bearbeitung an gewählten Positionen möglich

Komfortable Funktionen in java.io.*

- Lesen/Schreiben aus Datei: *FileInputStream/FileOutputStream*
- Zeichenbasierte Ein- und Ausgabe: *FileReader/FileWriter*
- Objektserialisierung: *ObjectInputStream/ObjectOutputStream*

Schreiben von Dateien mit FileWriter

Intention

- Ausgabe von alphanumerischen Zeichen in Dateien

  $\Rightarrow$ mögliche Konstruktoren

```
// Schreiben in Datei mit dem Bezeichner name
FileWriter (String name)
// append: neue Daten anhaengen oder ueberschreiben?
FileWriter (String name, boolean append)
// Schreiben in Datei file
FileWriter (File file)
```

Methoden (Auswahl)

```
// Schreiben von Daten
void write (String str)
 // Datei wird geschlossen
void close()
// Leeren von Puffern und Schreiben erzwingen
void flush()
```

Beschleunigtes Schreiben mit BufferedWriter

Problem

- Ausgabe auf externen Speichern (z.B. Festplatte) erzeugt Aufwand für Betriebssystem

  ⇒ *verlangsamte Ausführung* von Applikationen

Lösung

- Datenhaltung zunächst nur in einem Puffer (Hauptspeicher)
- wenn Puffer voll, erfolgt Schreibvorgang auf die Festplatte

  ⇒ *verbesserte Effizienz* durch weniger *write*-Aufrufe

Umsetzung

```
// Ausgabe wird an Writer uebergeben
BufferedWriter (Writer out)
// Initiale Puffergroesse wird festgelegt
BufferedWriter (Writer out, int size)
```