

Innere Klassen

Innere Klassen

- Java 1+ :-)
- Eine "Inner Class" wird innerhalb des Codeblocks einer anderen Klasse vereinbart
- Die bisher eingeführten Klassen werden auch Top-Level-Klassen genannt.
- Elegante?
- Nützliche?

Innere Klassen

- Definition von Hilfsklassen möglichst nahe an der Stelle, wo sie gebraucht werden

```
public class TopLevelClass1 {  
}
```

TopLevelClass.java

```
public class OuterClass {  
    . . .
```

OuterClass.java

```
    public class InsideClass {  
    }  
}
```

```
}
```

The Zoo of inner classes

- Elementklassen
 - innere Klassen, die in anderen Klassen definiert sind
- Geschachtelte Klassen
 - sind Top-Level-Klassen und Interfaces, die innerhalb anderer Klassen definiert sind
- Lokale Klassen
 - Klassen, die innerhalb einer Methode oder eines Java-Blocks definiert werden
- Anonyme Klassen
 - Lokale und namenlose Klassen

Geschachtelte Top-Level-Klassen

```
class EnclosingClass{  
    . . .  
    static class StaticNestedClass {  
        . . .  
    }  
    class InnerClass {  
        . . .  
    }  
} // end of enclosing class
```

Geschachtelte Top-Level-Klassen

```
public class A {  
    static int i = 4711;  
    public static class B {  
        int my_i = i;  
        public static class C { ... } //end of class C  
    } // end of class B  
} // end of class C
```

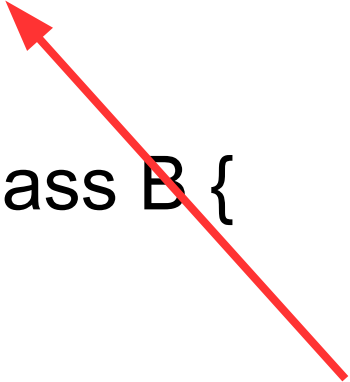
```
A a = new A();
```

```
A.B ab = new A.B();
```

```
A.B.C abc = new A.B.C();
```

Geschachtelte Top-Level-Klassen

```
public class A {  
    static String a = "A";  
    String b = "B";  
    public static class B {  
        void m() {  
            System.out.println(a);  
        }  
    } // end of class B  
} // end of class A
```



Elementklassen

- Elementklassen echte innere Klassen im Gegensatz zu den geschachtelten Top-Level-Klassen, die nur zur Strukturierung dienen
- Eine Elementklasse hat Zugriff auf alle Variablen und Methoden ihrer umgebenden Klasse
- Elementklassen werden analog gebildet und benutzt wie normale Klassen.

Elementklassen

```
public class A {  
    . . .  
    public class B {  
        . . .  
        public class C {  
            . . .  
        }  
    }  
}
```

javac A.java



A.class A\$B.class A\$B\$C.class

Elementklassen

- Objekte von Elementklassen sind immer mit einem Objekt der umgebenden Klasse verbunden

```
public class A {  
    public static int i = 30;  
    public class B {  
        int j = 4;  
        public class C {  
            int k = i;  
        }  
    }  
}
```

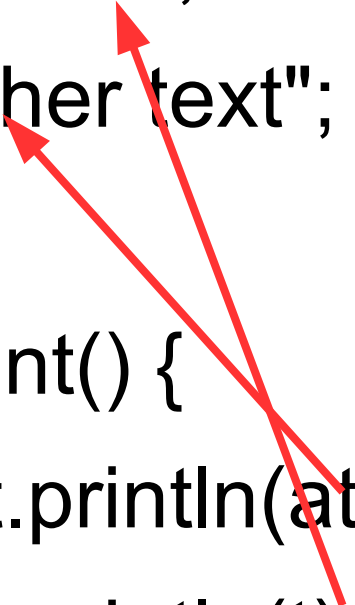
```
A a = new A();  
A.B b = a.new B();  
A.B.C c = b.new C();
```

Elementklassen

- Jeder Instanz einer Elementklasse ist ein Objekt der umgebenden Klassen zugeordnet.
- Damit kann das Objekt der Elementklasse implizit auf die Instanzvariablen der umgebenden Klasse zugreifen
- Elementklassen dürfen keine statischen Elemente

Elementklassen

```
public class H {  
    static String t = "text";  
    String at = "another text";  
    public class B {  
        public void print() {  
            System.out.println(at);  
            System.out.println(t);  
        }  
    }  
} // end of class B  
} // end of class H
```

Two red arrows originate from the 'print()' method in the inner class B. One arrow points to the 'at' variable in the line 'String at = "another text";', and the other points to the 't' variable in the line 'System.out.println(t);'.

Lokale Klassen

- Lokale Klassen sind innere Klassen, die nicht auf oberer Ebene in anderen Klassen verwendbar sind, sondern nur lokal innerhalb von Anweisungsblöcken von Methoden.

```
public class C {  
  
    ...  
  
    public void doSomething() {  
  
        int i = 0;  
  
        class X implements Runnable {  
  
            public X() {...}  
  
            public void run() {...}  
  
        }  
  
        new X().run();  
  
    } // end of doSomething  
  
}
```

Lokale Klassen

- Lokale Klassen dürfen folglich nicht als public, protected, private oder static deklariert werden
- Lokale Klassen dürfen keine statischen Elemente haben
- Eine Lokale Klasse kann im umgebenden Codeblock nur die mit final markierten Variablen und Parameter benutzen

Lokale Klassen

```
public class H {  
    String t = "text";  
  
    public void m() {  
        final String mt = "in m";  
  
        class C {  
            void h() {  
                System.out.println(t); // Instanzvariable  
                System.out.println(mt); // mt ist final  
            }  
        } // end of class C  
  
        C in_m = new C();  
        in_m.h();  
    } // end of method m  
  
    public static void main( String[] args ) {  
        H h = new H();  
        h.m();  
    }  
}
```

Anonyme Klassen

- haben keinen Namen
- haben keinen Konstruktor
- sie entstehen immer zusammen mit einem Objekt
- werden wie lokale Klassen innerhalb von Anweisungsblöcken definiert

`new-expression class-body`

Anonyme Klassen

```
abstract class Person{
    abstract void eat();
}

class TestAnonymousInner{

    public static void main(String args[]){

        Person p=new Person(){
            void eat(){
                System.out.println("nice fruits");
            }
        };

        p.eat();
    }
}
```

Ausnahmebehandlung

Fehlerhafte Programme

- Ein Programm kann aus vielen Gründen unerwünschtes Verhalten zeigen.
- Fehler beim Entwurf
- Fehler bei der Programmierung des Entwurfs
 - Algorithmen falsch implementiert
- Ungenügender Umgang mit außergewöhnlichen Situationen
 - Abbruch der Netzwerkverbindung
 - Dateien können nicht gefunden werden
 - fehlerhafte Benutzereingaben

Umgang mit außergewöhnlichen Situationen

Ausnahmesituationen unterscheiden sich von Programmierfehlern darin, dass man sie nicht (zumindest prinzipiell) von vornherein ausschließen kann.

Immer möglich sind zum Beispiel:

- unerwartete oder ungültige Eingaben
- Ein- und Ausgabe-Fehler beim Zugriff auf Dateien oder Netzwerk

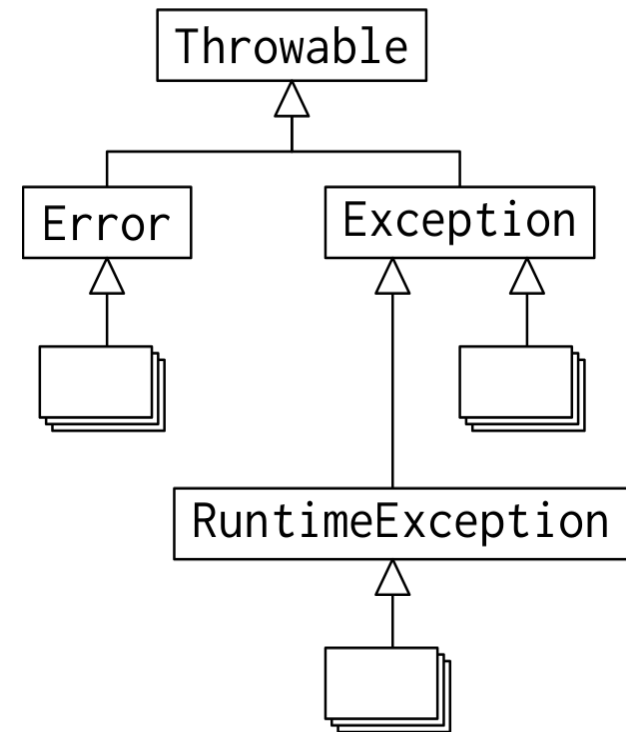
Ausnahmen in Java

- Die Erkennung und die Behandlung eines Fehlers muss oft in ganz verschiedenen Teilen des Programms stattfinden.
- Beispiel: Datei öffnen
 - Erkennung: InputStream
 - Behandlung: GUI
- Exceptions sind ein Mechanismus, um bei der Erkennung eines Fehlers eine Ausnahme auszulösen, die anderswo behandelt werden kann.
- Ohne Exceptions?

Ausnahmen in Java

In Java werden verschiedene Arten von Ausnahmen durch verschiedene Unterklassen von Throwable repräsentiert.

- Instanzen von Error
- Instanzen von Exception
- Instanzen von RuntimeException



Auslösen von Ausnahmen

- Das Auslösen einer Ausnahme erfolgt mit der Anweisung *throw exp*;
wobei *exp* Ausdruck vom Typ *Throwable* ist.
- Versuche existierende Ausnahmeklassen zu verwenden:
- `NullPointerException`
- `IllegalArgumentException`
 - Eine Methode wurde mit unzulässigen Parametern aufgerufen
- `IllegalStateException`

Deklaration von Ausnahmen

- Für Methoden kann man möglicherweise auftretende Ausnahmen deklarieren.

```
public void m() throws IOException {  
    if (...) {  
        throw new IOException();  
    }  
}
```

// Annotationen sagen nur, dass eine Ausnahme möglicher-
// weise auftritt. Tatsächlich kann sie auch nie auftreten.

```
public void n() throws IOException {  
    System.out.println();  
}
```


Behandlung von Ausnahmen

```
try {  
    // Block fuer "normalen" Code  
} catch (Exception1 e) {  
    // Ausnahmebehandlung fuer Instanzen von Exception1  
} catch (Exception2 e) {  
    // Ausnahmebehandlung fuer Instanzen von Exception2  
} finally {  
    // Code, der in jedem Fall nach normalem  
    // Ausnahmebehandlung ausgefuehrt werden  
}
```

Checked Exceptions

Geprüfte Ausnahmen repräsentieren Ausnahmesituationen, mit denen das Programm rechnen kann und auf die es reagieren sollte.

- Geprüfte Ausnahmen sind alle Unterklassen von `Exception`, die nicht auch Unterklassen von `RuntimeException` sind.
- Beispiel: `FileNotFoundException`, `IOException`
- Geprüfte Ausnahmen müssen entweder behandelt werden oder als möglich deklariert werden.

Unchecked Exceptions

Ungeprüfte Ausnahmen repräsentieren Ausnahmesituationen, deren Ursache ein Programmierproblem ist.

- Alle Ausnahmeklassen, die von `RuntimeException` abgeleitet sind, sind ungeprüfte Ausnahmen.
- Beispiele: `NullPointerException`, `IllegalArgumentException`
- Ungeprüfte Ausnahmen müssen weder behandelt noch deklariert werden.

Fehlerbehandlung

Ein fertiges Programm sollte nie mit einer Exception abbrechen.

- Geprüfte Exceptions sind an einer geeigneten Stelle mit try abzufangen und zu behandeln
- Ausnahmesituationen müssen sinnvoll behandelt werden
- falsche Benutzereingabe -> neue Eingabeaufforderung
- IO-Fehler -> nochmal versuchen
- nicht sinnvoll behandelbarer Fehler
 - > Benutzerdaten sichern, Programm beenden

Fehlerbehandlung

Ungeprüfte Ausnahmen, die Programmierfehler repräsentieren, werden nicht abgefangen.

- `NullPointerException`
- `IllegalArgumentException`
- `ClassCastException`

Die einzige sinnvolle Reaktion auf solche Exceptions ist das Programm zu korrigieren.

- Kein Abfangen solcher Exceptions mit try.

Konvention

Öffentliche (public) Methoden überprüfen eventuelle Annahmen an ihre Parameter und lösen gegebenenfalls eine Exception aus.

```
/**
 * Konstruiere eine neue Node mit den gegebenen Daten
 *
 * @param id eindeutiger Name der Node, nicht null
 * @param latitude Koordinate
 * @param longitude Koordinate
 */
public Node(String id, double latitude, double longitude) {
    if (id == null) {
        throw new NullPointerException();
    }
    this.id = id;
    this.longitude = longitude;
    this.latitude = latitude;
}
```

Fehlerbehandlung

```
try {  
    ...  
} catch (IOException e) {  
    ...  
} catch (JSONException e) {  
    ...  
}
```

Fehlerbehandlung

besser als ein catch everything

```
try {  
    ...  
} catch (Exception e) {  
    ...  
}
```

```
try {  
    ...  
} catch (Exception e) { }
```


Fehlerbehandlung

Nope:

```
try {  
    Iterator<String> i = list.iterator();  
    while (true) {  
        String s = i.next();  
        ...  
    }  
} catch (NoSuchElementException e) { }
```

Yes:

```
for (String s: list) {  
    ...  
}
```

Dokumentation

- Ungeprüfte Exceptions werden üblicherweise nicht mit throws deklariert.
- Die möglichen ungeprüften Exceptions sollten jedoch im Javadoc dokumentiert werden.

```
/**  
 * Returns the element at the specified position in this list.  
 *  
 * @param index index of the element to return  
 * @return the element at the specified position in this list  
 * @throws IndexOutOfBoundsException {@inheritDoc}  
 */
```

Hinweise

- Ungeprüfte Ausnahmen, die Programmierfehler repräsentieren, nicht abfangen
- Argumente in öffentlichen Methoden überprüfen
- Ausnahmen möglichst spezifisch behandeln
- Ausnahmen nicht ignorieren
- Ausnahmen nur in außergewöhnlichen Situationen verwenden
- Ausnahmen dokumentieren

JSON

Datenaustausch

Kodierung von Daten

- Binärformate (PNG, MP4, Word, . . .)
effizient, aufwändig, nicht menschenlesbar
- Textformate (Java, . . .):
menschenlesbar, Aufwand für Ein- und Ausgabe
- generische Formate (XML, JSON, . . .):
Datenaustausch, implementiert in Bibliotheken

Datenaustausch

- JSON (JavaScript Object Notation)
- einfaches textbasiertes Datenaustauschformat
- menschenlesbar
- Standardisiert in RFC 4627

JSON

- “Objekte” mit Attribut:Wert-Zuordnungen
- Leerzeichen außerhalb von Strings, Zeilenumbrüche nicht relevant

```
{
  "type": "node",
  "id": "363179",
  "lat": 48.1408871, "long": 11.5615991
},
{
  "type": "way", "id": "372802991",
  "nd": ["3763512880", "3763512881", "1545920068"],
  "tags": {
    "bus": "yes",
    "name": "Herkomerplatz",
    "highway": "platform"
  }
}
```


Datentypen

- Standard-Datentypen
 - Strings in Anführungszeichen, Escaping mit \ (wie in Java)
 - Zahlen (z.B. -12, 12E9, 12.9)
 - Boolesche Werte (true, false)
 - Null-Wert durch Schlüsselwort null
- Arrays
 - In eckigen Klammern (z.B. [1,2,3,4])
- Objekte
 - in geschweiften Klammern
 - Attribute durch Strings benannt

Verarbeitung von JSON-Daten

Es gibt viel Bibliotheken zur Ein- und Ausgabe von JSON:

- org.json
 - für Java auf <http://json.org> verfügbar
- Jackson
- GSON

Parsing

```
{ "type": "way", "nd": ["3763512880", "3763512881", "1545920068"], "tags":  
{ "name": "Herkomerplatz", "highway": "platform" } }
```

```
String s =
```

```
JSONObject json = new JSONObject (s);
```

```
String t = json.getString ("type"); // " way "
```

```
JSONArray nd = json.getJSONArray ("nd");
```

```
double d1 = nd.GetDouble (1); //3763512881
```

Parsing

```
{ "type": "way", "nd": ["3763512880", "3763512881", "1545920068"], "tags":  
{ "name": "Herkomerplatz", "highway": "platform" } }
```

```
String s =
```

```
JSONObject json = new JSONObject (s);
```

```
JSONObject tags = json.getJSONObject ("tags");
```

```
String n = tags.getString ("name ");
```

```
for ( String k: json.keySet ())
```

```
System.out.println (k );
```

Ausgabe von JSON-Daten

```
JSONObject json = new JSONObject ();  
json.put ("type" , "node");  
json.put ("id", "34");  
json.put ("lat" , 31.3);  
json.put ("long" , 12.8);  
System.out.println (json);
```

Ausgabe:

```
{"id":"34","type":"node","lat":31.3,"long":12.8}
```

JSON

- JSON dient nur zum Datenaustausch.
z.B. JSONObject nicht zur Datenrepräsentation.
- Beim Austausch von Text ist auf die Textkodierung zu achten.
- Behandeln Sie bei der Programmierung alle möglichen Fehlerfälle. JSON-Daten, die aus einer Datei gelesen werden, können nicht als wohlgeformt angenommen werden.

JAVA Ein/Ausgabe

IO

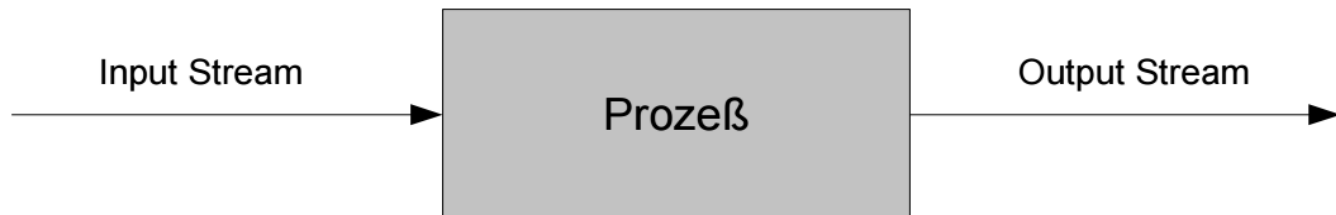
- Package java.io
- Datenströme
- Input-Streams
- Output-Streams

Ein- und Ausgabe in Java

- Einlesen und Ausgeben von Dateien
- Ausgabe auf dem Bildschirm
- Einlesen von der Tastatur
- Beinahe alle IO-Methoden können eine Exception werfen
- Die meisten Exceptions sind vom Typ `java.io.IOException`

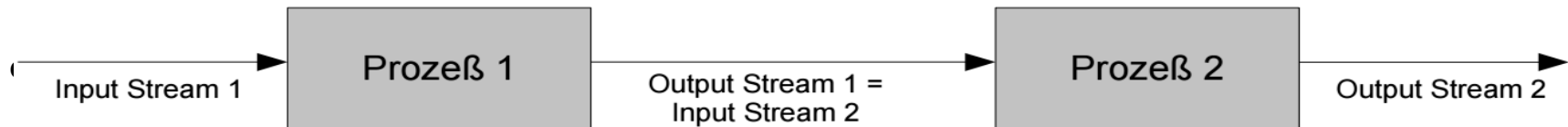
Datenströme

- Input Stream
 - Ein Daten-Strom, der von einer DatenQuelle zum verarbeitenden Prozess führt
 - Tastatur
 - File System
- Output Stream
 - Ein Daten-Strom, der vom Computer zu einer Daten-Senke führt
 - Bildschirm
 - Drucker
 - File System

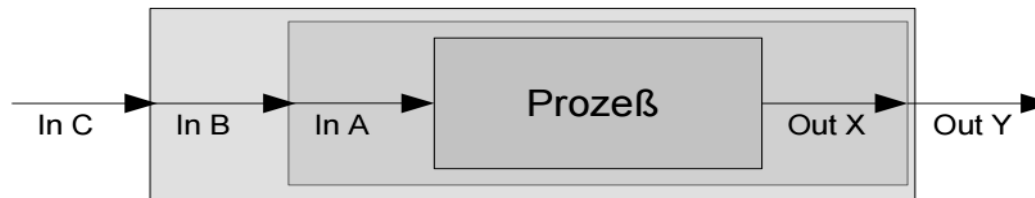


Datenströme

- Datenströme können beliebig miteinander kombiniert werden



- Schachteln von Streams
 - am Eingabeteil wird ein Vorverarbeitungsschritt vorgeschaltet
 - am Ausgabeteil wird eine Nachverarbeitung durchgeführt
 - das erlaubt das Konstruieren von abstrakteren Streams auf der Basis von einfachen Streams



Byte und Character Streams

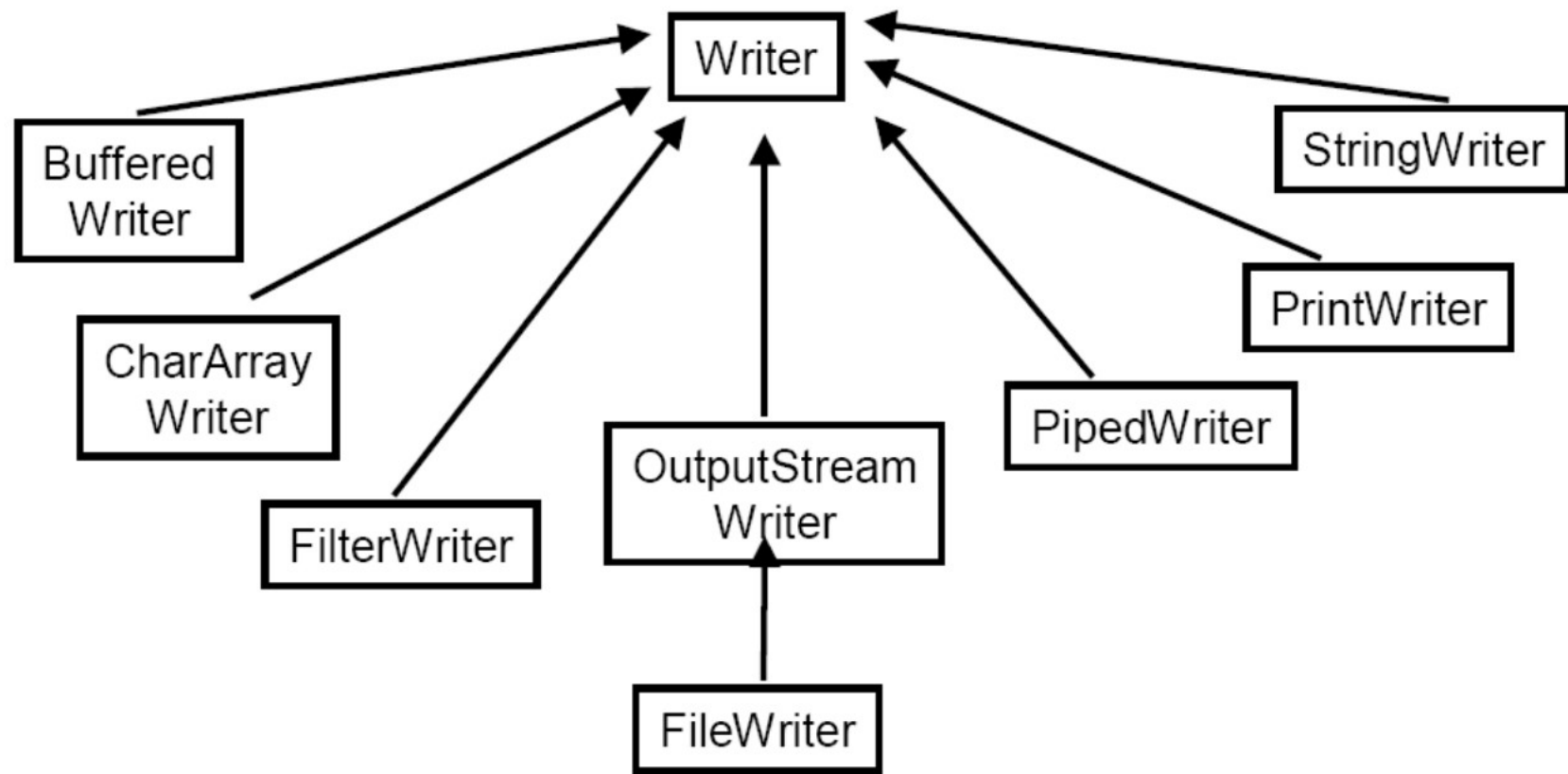
- Zwei grundlegende Typen von Streams:
 - Byte-Streams
 - Übertragen wird nur ein einzelnes Byte (8 bit)
 - Character-Streams
 - Übertragen wird ein ganzes Zeichen (in Java 16 bit, Unicode)

java.io.Writer

Abstrakte Basisklasse für alle Character Output-Streams

- `public void close()`
- `public void write(int b) throws IOException`
- `public void write(String s) throws IOException`
- `public void write(String s, int start, int n) throws IOException`

Überblick über Writer-Klassen



Buffering

- In vielen Fällen wird nach einem write nicht sofort geschrieben
- sondern es wird gewartet, bis sich eine gewisse Menge von Daten angesammelt haben
- die werden dann in regelmäßigen Abständen automatisch geschrieben
- Mit Hilfe der flush-Methode kann man das Schreiben erzwingen
- `public void flush()`
 - schreiben aller noch ausstehenden Daten

java.io.FileWriter

- `public FileWriter(String name) throws IOException`
 - Öffnet das File mit dem Namen `name` zum Schreiben
 - Falls das Öffnen des Files schiefgeht, wirft die Methode eine `IOException`
- `public FileWriter(String n, boolean app) throws IOException`
 - öffnet das File mit Namen `n` zum Schreiben
 - Falls die boolsche Variable `app` auf `true` gesetzt ist, wird an das File angehängt

java.io.FileWriter

```
1  import java.io.*;
2  public class WriteToFile
3  {
4      public static void main(String[] args)
5      {
6          FileWriter out;
7          try {
8              out = new FileWriter("hallo.txt");
9              out.write("Hallo JAVA\r\n");
10             out.close();
11         }
12         catch (Exception e) {
13             System.err.println(e.toString());
14             System.exit(1);
15         }
16     }
17 }
18
```

java.io.StringWriter

- Ein String kann ebenso als Ausgabe-Einheit betrachtet werden wie ein File
- implementiert alle Methoden von Writer
- toString()
- GetBuffer()
- Analog dazu gibt es die Klasse CharArrayWriter

Schachteln von Streams

- Manche Methoden verwenden einen bereits definierten Stream
- `FilterWriter`
 - Abstrakte Basisklasse für die Konstruktion von Ausgabefiltern
- `PrintWriter`
 - Ausgabe aller Basistypen im Textformat
- `BufferedWriter`
 - `Writer` zur Ausgabepufferung

java.io.PrintWriter

- Dient zur Ausgabe von Texten
- `print()`
- Es gibt eine `print`-Methode für jeden Standard-Typ
- `println()`
- `System.out` ist eine Klassen-Konstante vom Typ `PrintStream`
- nur für Byte-Streams statt Character-Streams

Beispiel

```
1 public static void main(String[] args)
2 {
3     PrintWriter pw;
4     double sum = 0.0;
5     int nenner;
6     try {
7         FileWriter fw = new FileWriter("zwei.txt");
8         BufferedWriter bw = new BufferedWriter(fw);
9         pw = new PrintWriter(bw);
10        for (nenner = 1; nenner <= 1024; nenner++ ) {
11            sum += 1.0 / nenner;
12            pw.print("Summand: 1/");
13            pw.print(nenner);
14            pw.print(" Summe: ");
15            pw.println(sum);
16        }
17        pw.close();
18    }
19    catch (IOException e) {
20        System.out.println("Fehler beim Erstellen der Datei");
21    }
22 }
```

Beispiel

```
1 public static void main(String[] args)
2 {
3     PrintWriter pw;
4     double sum = 0.0;
5     int nenner;
6     try {
7         pw = new PrintWriter(
8             new BufferedWriter(
9                 new FileWriter("zwei.txt") ) );
10        for (nenner = 1; nenner <= 1024; nenner++ ) {
11            sum += 1.0 / nenner;
12            pw.print("Summand: 1/");
13            pw.print(nenner);
14            pw.print(" Summe: ");
15            pw.println(sum);
16        }
17        pw.close();
18    }
19    catch (IOException e) {
20        System.out.println("Fehler beim Erstellen der Datei");
21    }
22 }
23
```

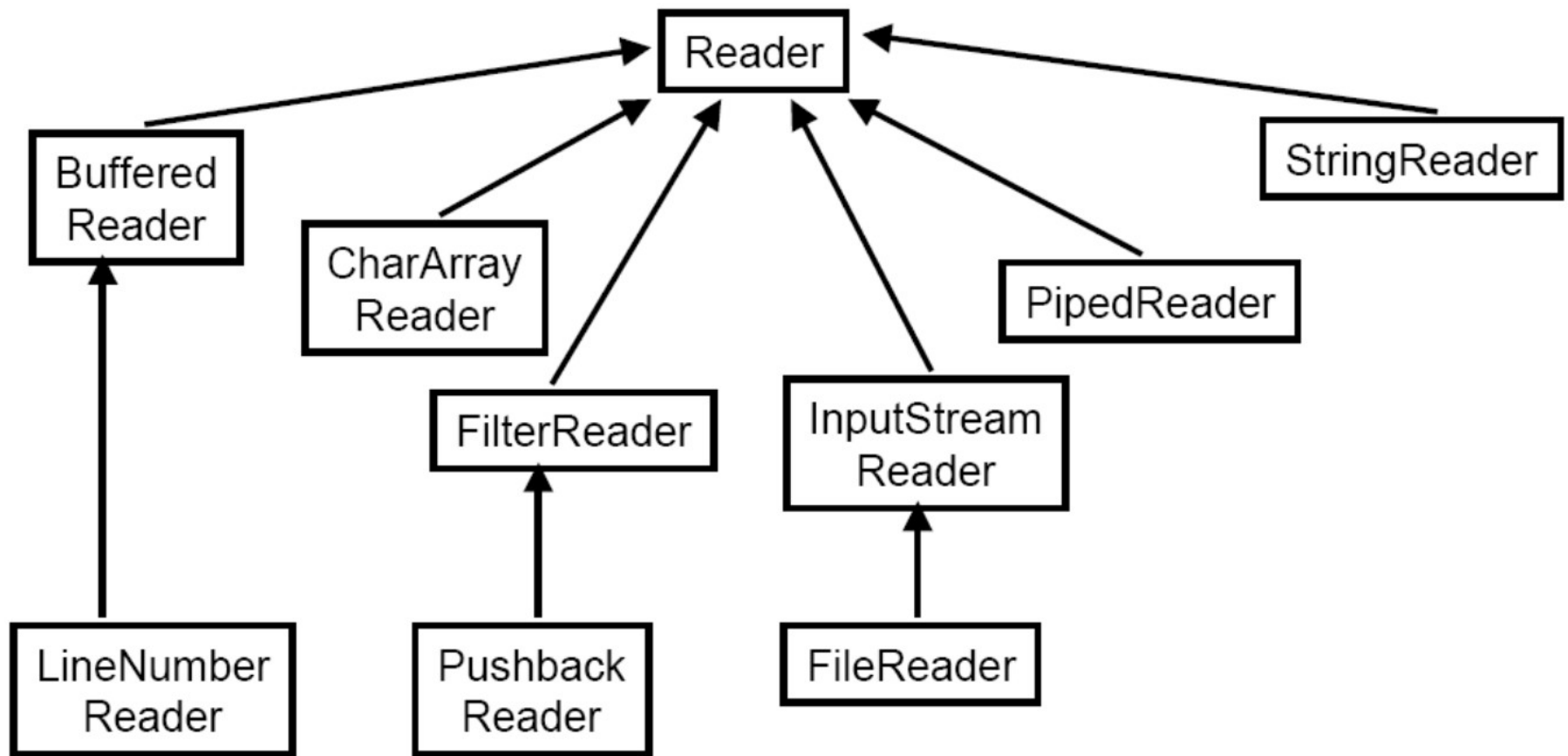
java.io.FilterWriter

- Abstrakte Klasse zur Definition eines Output-Filters
- Konstruktor benötigt daher wiederum einen existierenden Output-Filter
- Es gibt keine vorgeschriebenen zusätzlichen Methoden

java.io.Reader

- Abstrakte Basisklasse für alle Character Input-Streams
- `public void close()`
- `public int read()` throws `IOException`
- `public int read(char[] c)` throws `IOException`
- `public int read(char[] c, int start, int n)`
throws `IOException`

Reader-Klassen



Beispiel

```
public static void main (String [] args) throws IOException {  
    Reader r = new FileReader ("dat.txt");  
    BufferedReader br = new BufferedReader(r);  
    try {  
        String line = br.readLine();  
  
        while (line != null) {  
            System.out.println(line);  
            line = br.readLine();  
        }  
    } finally {  
        // Streams sollten immer geschlossen werden, am besten  
        // in einem finally - Block.  
        br.close();  
    }  
}
```

Objektserialisierung

Serialisierung von Objekten

- Umwandlung des Objektzustandes in einen Strom von Bytes, aus dem eine Kopie des Objekts zurückgelesen werden kann
- Java: einfacher Mechanismus zur Serialisierung von Objekten
- eigenes Datenformat
- Speicherung von internen Programmzuständen
- Übertragung von Objekten zwischen verschiedenen JVMs
- JSON??

Serialisierung in Java

Ablauf der Serialisierung eines Java-Objekts:

- Metadaten, wie Klassenname und Versionsnummer, in den Byte-Strom schreiben
- alle nichtstatischen Attribute (private, protected, public) serialisieren
- die entstehenden Byte-Ströme in einem bestimmten Format zu einem zusammenfassen

Serialisierung in Java

- Kennzeichnung von serialisierbaren Objekten:
Klasse implementiert das Interface
`java.io.Serializable`
- Attribute einer serialisierbaren Klasse sollten
Basistypen oder serialisierbare Objekte sein
- Gründe für Kennzeichnungspflicht:
 - Sicherheit (private Attribute)
 - Serialisierbarkeit soll aktiv vom Programmierer
erlaubt werden

Serialisierung von Objekten

- Objekte schreiben

```
FileOutputStream f = new FileOutputStream("datei");  
ObjectOutput s = new ObjectOutputStream(f);  
s.writeObject(new Integer(3));  
s.writeObject("Text");  
s.flush();
```

- Objekte lesen

```
FileInputStream in = new FileInputStream("datei");  
ObjectInputStream s = new ObjectInputStream(in);  
Integer int = (Integer)s.readObject();  
String str = (String)s.readObject();
```

Transient

- Attribute, die nicht serialisiert werden sollen, können als transient markiert werden
- Caches
- nichtserialisierbare Felder
- sensitive Daten

```
public class Account {  
    private String username;  
    private transient String password;  
}
```


Anpassen der Serialisierungsprozedur

- Serialisierungsmethoden können angepasst werden
- Schreiben von zusätzlichen Daten
- wiederherstellen von transienten und nichtserialisierbaren Feldern
- Dazu müssen in der serialisierbaren Klasse zwei Methoden mit folgender Signatur geschrieben werden:

```
private void writeObject(ObjectOutputStream oos) throws IOException
```

```
private void readObject(ObjectInputStream ois)
```

```
throws ClassNotFoundException, IOException
```

Beispiel

```
private void writeObject(ObjectOutputStream oos)
    throws IOException {
    // zuerst die Standardserialisierung aufrufen:
    oos.defaultWriteObject();
    // zusätzliche Daten schreiben
    oos.writeObject(new java.util.Date());
}

private void readObject(ObjectInputStream ois)
    throws ClassNotFoundException, IOException {
    // zuerst die Standarddeserialisierung aufrufen:
    ois.defaultReadObject();
    // zusätzliche Daten lesen:
    date = (Date)ois.readObject();
    // mit transient markierte Felder wiederherstellen
    ...
}
```

Versionsnummern

- Serialisierte Objekte haben eine Versionsnummer. Objekte mit falscher Versionsnummer können nicht deserialisiert werden
- Versionsnummer kann als statisches Attribut definiert werden:

```
public static long serialVersionUID = 1L
```
- Ist keine Nummer angegeben, so benutzt Java einen Hashwert