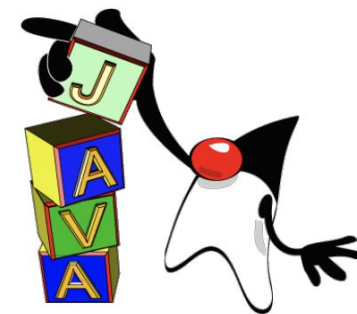
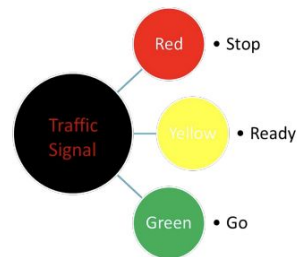


Ein- und Ausgabe in Java





Inhalt

- Maven
- JSON Beispiel
- Ein- und Ausgabe Funktionen
- Serialisierung



Anforderungen in Development

- Sourcecode bauen und packen
- Testen
 - Automatische Ausführen von Tests
 - Von Unit bis zu Akzeptanz-Tests
 - Codeanalyse
- Dokumentation
 - JavaDoc
 - SourceCode
 - Komplette Projekt-Dokumentation
- Dependency-Management

<https://mvnrepository.com>

Anforderungen in Development

- Verteilung der erzeugten Artefakte
- Konfigurations-Management
 - Programme releasen
 - Programm-Versionen sauber verwalten
- Dependency

Ant

Maven

Gradle

Automatisierung

Wenig menschl.
Interaktion

Geschwindigkeit



Maven

- Deklarativer Ansatz
- Feste Lifecycle für Aufräumen, Bauen und Dokumentation
- Funktionalität wird durch Plugins bereitgestellt
 - Plugins werden an Phasen im Lifecycle gebunden
 - Plugins auch für Release- und Dokumentations-Prozesse
- Convention over Configuration
 - Vorgaben für alle Details eines Builds: Verzeichnisse, Ablauf, Distribution
 - Nur wer von Vorgaben abweichen möchte, muss die Konfiguration anpassen
- Dependency-Management
 - Auch Maven-Plugins sind Abhängigkeiten



Archetypes – Projekt-Templates

- Es sind Projekt-Templates für typische Projektarten definiert
- Diese Templates installieren Basisversionen, die dann weiterentwickelt werden
- Es gibt sehr viele archetypes, z.B. für Web-Anwendungen
- Zur Erzeugung wird das Plugin archetype benutzt:
`mvn archetype:generate`

Beispiel

```
(01) <project xmlns="http://maven.apache.org/POM/4.0.0"
(02)   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(03)   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
(04)   http://maven.apache.org/xsd/maven-4.0.0.xsd">
(05)   <modelVersion>4.0.0</modelVersion>
(06)   <groupId>de.hallo</groupId>
(07)   <artifactId>hello</artifactId>
(08)   <version>1.0-SNAPSHOT</version>
(09)   <packaging>jar</packaging>
(10)   <name>hello</name>
(11)   <url>http://maven.apache.org</url>
(12)   <properties>
(13) <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
(14)   </properties>
(15)   <dependencies>
(16)     <dependency>
(17)       <groupId>junit</groupId>
(18)       <artifactId>junit</artifactId>
(19)       <version>3.8.1</version>
(20)       <scope>test</scope>
(21)     </dependency>
(22)   </dependencies>
(23) </project>
```

Maven-Konfiguration

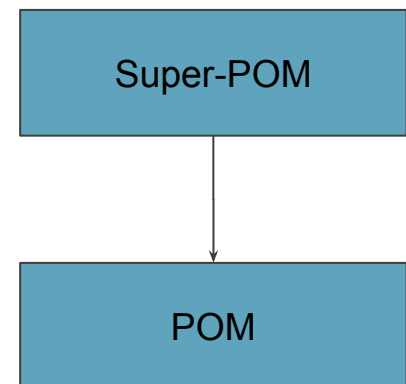
- Die Konfiguration erfolgt durch eine XML-Datei
- Nur eine XML-Datei pro Projekt: pom.xml (Project Object Model)
- Keine Möglichkeit von XML-Importen
- Einstellungen werden vom Eltern-Projekt übernommen
- Es gibt eine Super-POM-Datei, von der alles geerbt wird

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>jtest</artifactId>
  <version>1.0-SNAPSHOT</version>

</project>
```



Maven

validate – Prüfung auf gültige und vollständige Projektstruktur

compile – Übersetzen

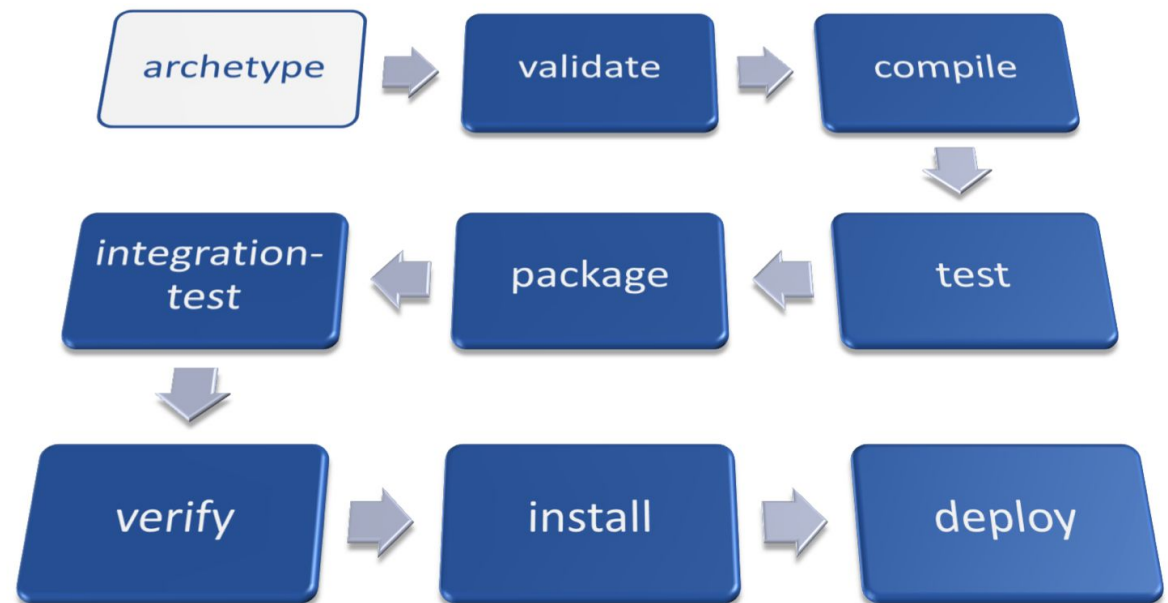
test – Durchführen der Tests

package – Erstellen der erzeugten Pakete

verify – Prüfung der Pakete

install – Installieren im lokalen Repository

deploy – Installieren im globalen Repository



JSON

- Jackson
- maven Projekt
- mapper Objekt
- manuell:

```
Reader reader = new BufferedReader(new  
    FileReader(...));
```

```
mapper.readTree(reader)
```

- automatisch:

```
mapper.readValue(new File(...), type...)
```



JSON - Jackson - Maven Repo

```
<dependency>  
  <groupId>com.fasterxml.jackson.core</groupId>  
  <artifactId>jackson-databind</artifactId>  
  <version>2.13.4.2</version>  
</dependency>
```



Maven - Not Supported Error

```
<properties>  
  <maven.compiler.source>1.8</maven.compiler.source>  
  <maven.compiler.target>1.8</maven.compiler.target>  
</properties>
```

Beispiel

```
public class Book {  
  
    private String title;  
    private String isbn;  
    private long year;  
    private String[] authors;  
  
    public Book() {  
    }  
  
    public Book(String title, String isbn,  
long year, String[] authors) {  
        this.title = title;  
        this.isbn = isbn;  
        this.year = year;  
        this.authors = authors;  
    }  
  
    // getters and setters, equals(),  
toString() .... (omitted for brevity)  
}
```

```
try {  
    // create object mapper instance  
    ObjectMapper mapper = new ObjectMapper();  
  
    // convert JSON string to Book object  
    Book book =  
mapper.readValue(Paths.get("book.json").toFile(  
    ), Book.class);  
  
    // print book  
    System.out.println(book);  
  
} catch (Exception ex) {  
    ex.printStackTrace();  
}
```



Java IO

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



Java.IO

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

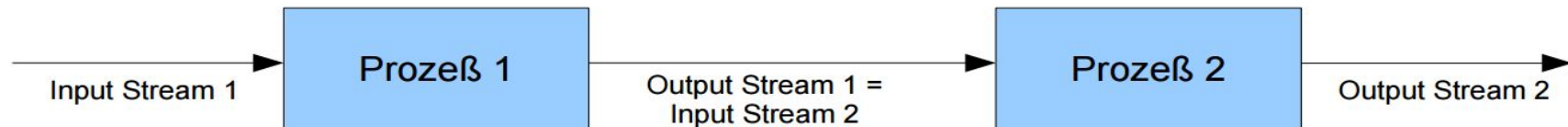
Datenströme

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

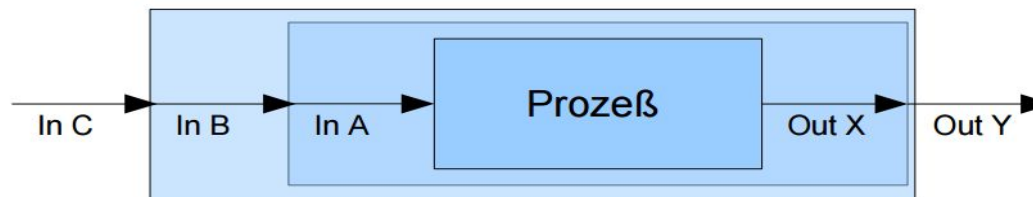


Datenströme

- Datenströme können beliebig miteinander kombiniert werden
 - Aneinanderhängen von Streams



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





Byte und Character Streams

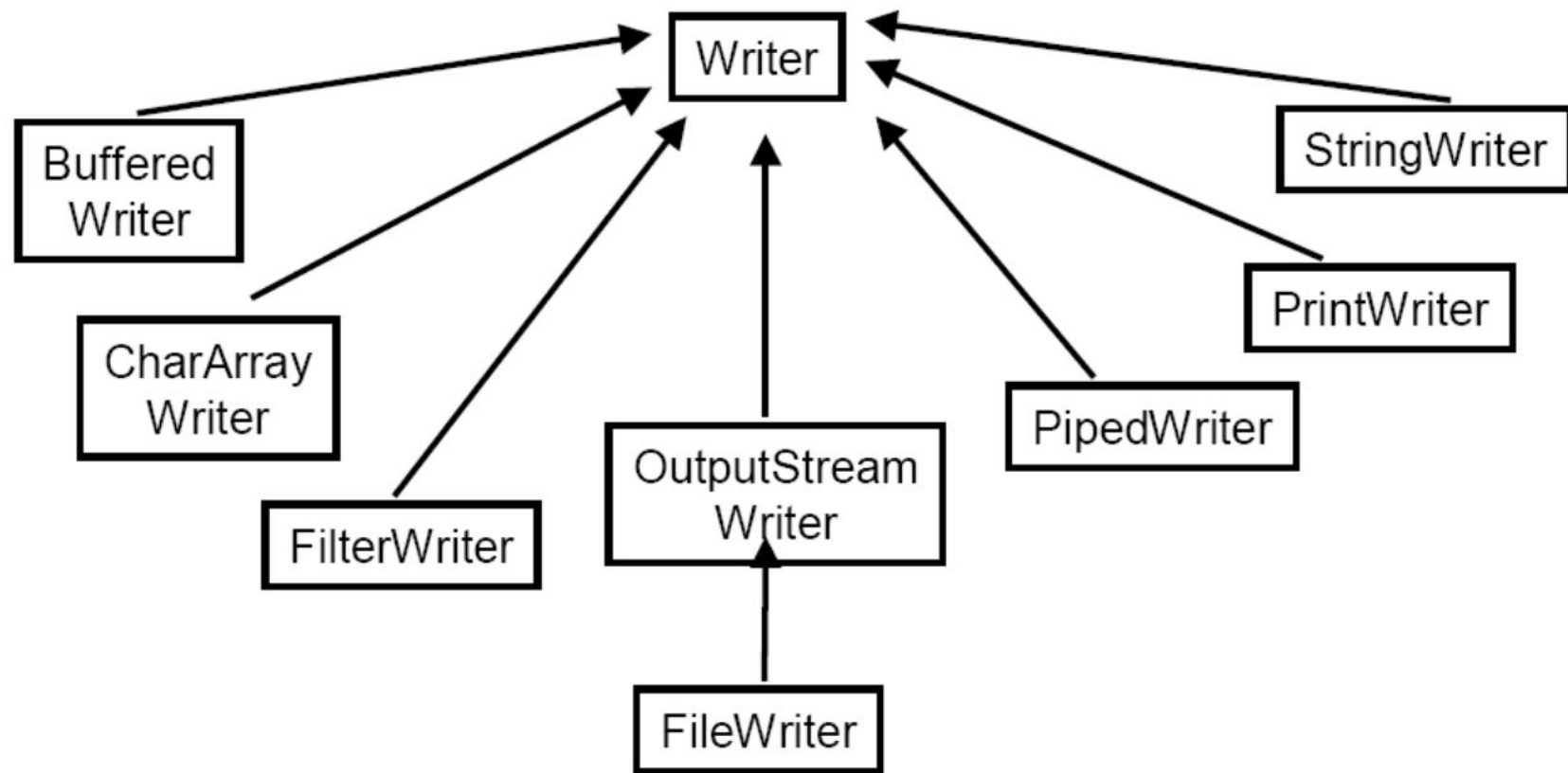
- Zwei grundlegende Typen von Streams:
 - Byte-Streams
 - Übertragen wird nur ein einzelnes Byte (8 bit)
 - Character-Streams
 - Übertragen wird ein ganzes Zeichen (in Java 16 bit, Unicode)



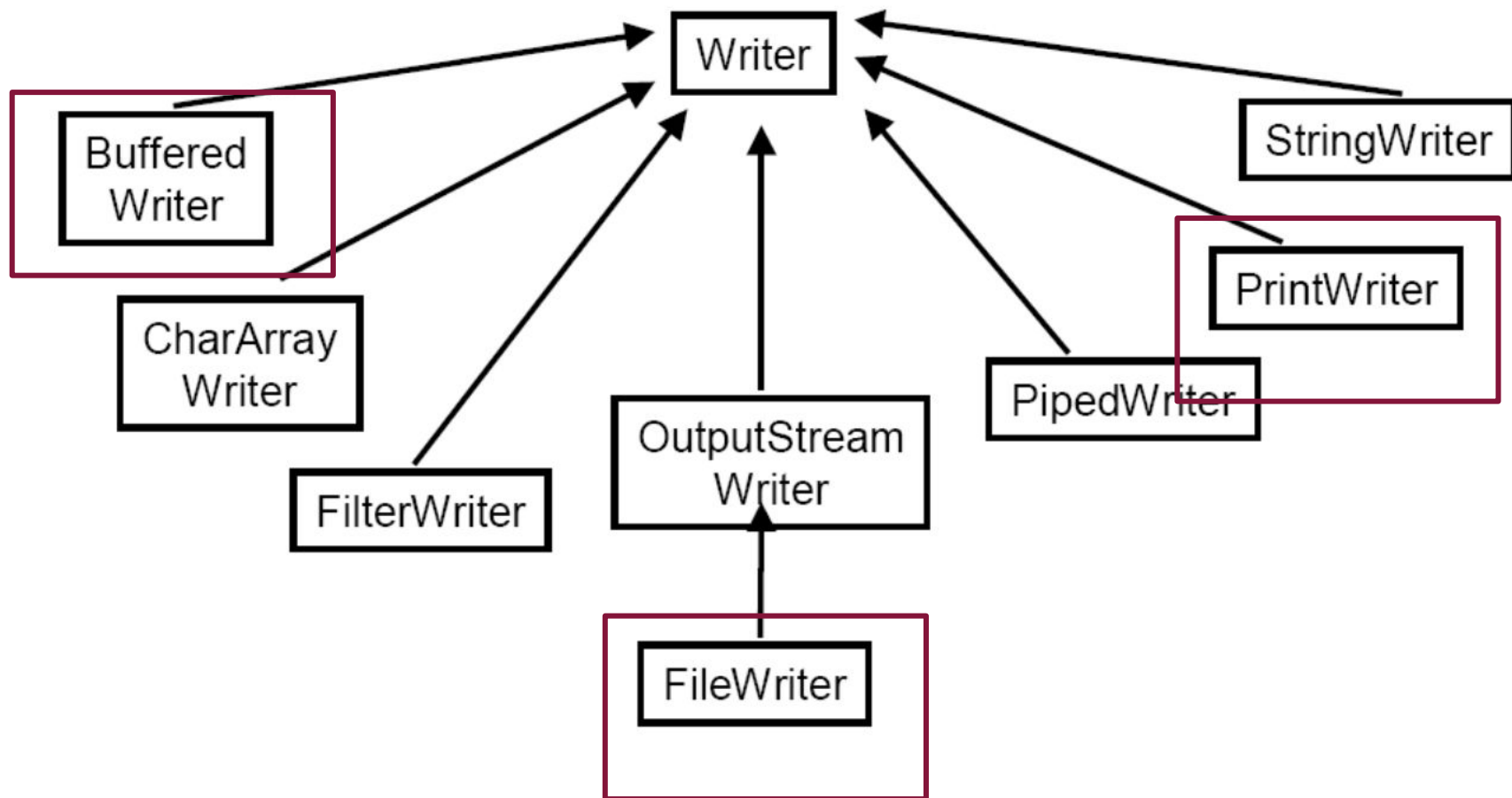
java.io.Writer

- 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



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 ausständigen Daten



java.io.FileWriter

- `public FileWriter(String name) throws IOException`
 - Öffnet die Datei mit dem Namen `name` zum Schreiben
 - Falls das Öffnen schlägt fehl, wirft die Methode eine `IOException`
- `public FileWriter(String n, boolean app) throws IOException`
 - 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 Filtern
- `PrintWriter`
 - Ausgabe aller Basistypen im Textformat
- `BufferedWriter`
 - Writer zu einer Output-Puffer



java.io.PrintWriter

- Dient zur Ausgabe von Texten
- `print()`
- Es gibt eine `print`-Methode für jeden Standard-Typ
 - `println()`
- `System.out` ist eine 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

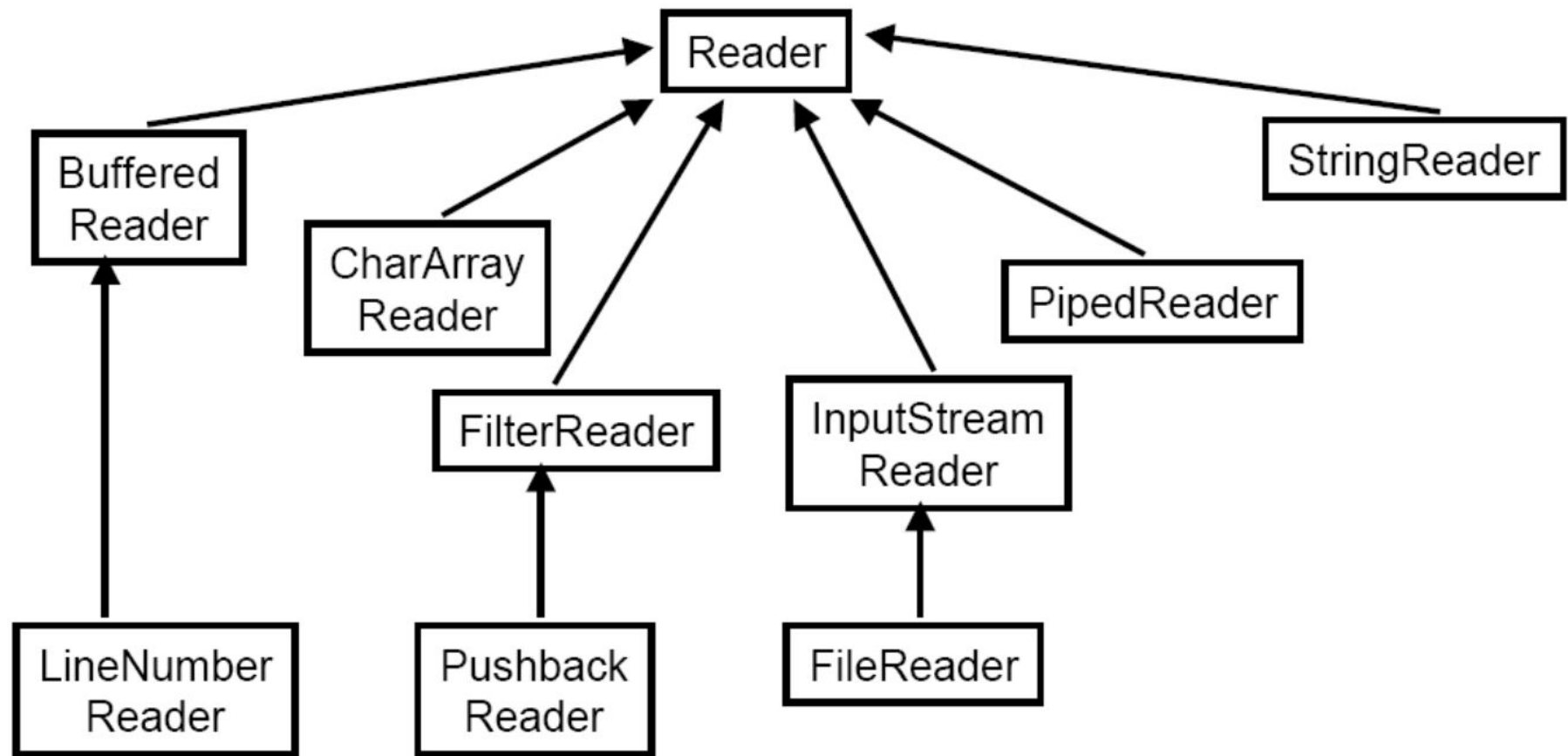
- abstrakte Klasse zur Definition eines Output-Filters
- Konstruktor benötigt daher wiederum einen existierenden Output-Filter
- es gibt keine vorgeschriebenen zusätzlichen Methoden



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();  
    }  
}
```



Serialisierung von Objekten

- Umwandlung des Objekts Zustandes 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 Bytestrom schreiben
- alle nicht statischen Attribute (private, protected, public) serialisieren
- die entstehenden Byteströme 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
 - nicht nur private Attribute
 - **Serialisierbarkeit soll aktiv vom Programmierer erlaubt werden**



Serialisierung von Objekten

- schreiben

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

- 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
- nicht serialisierbare 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 nicht serialisierbaren 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



Serialization

- Probleme mit diesem Ansatz
- JSON