



Java Einführung

IO (Eingabe/Ausgabe)

Inhalt dieser Einheit



Ein-/Ausgabe:

- Arbeiten mit Verzeichnissen und Dateien
- Schreiben und Lesen von Dateien bzw. Datenströmen
- Fehlerbehandlung beim Schreiben und Lesen von Dateien bzw. Datenströmen

Lesen und Schreiben von Daten

„Ein Programm muss oft Informationen aus einer externen Quelle **importieren (Datenquelle)** oder Information in eine externe Quelle **exportieren (Datensenke)**.“

„Die Information kann sich in einer **Datei** auf einer **Diskette**, irgendwo im **Netzwerk**, im **Speicher**, in einem anderen **Programm** befinden oder über die **Tastatur** eingegeben werden.“

„Die Information kann verschiedenen Typs sein, zum Beispiel **Objekte**, **Zeichen**, **Bilder** oder **Sounds**.“

(Quelle: <http://java.sun.com>)

In dieser Einheit werden Klassen vorgestellt, die ein Java Programm benötigt, um Informationen zu lesen und zu schreiben.

Files & Unterverzeichnisse

Für die Verwendung der Ein-Ausgabe-Klassen muss das Package `java.io` importiert werden.

Dateien und Unterverzeichnisse sind vom Datentyps `File` aus dem Package `java.io`:

- Datei:

```
File f = new File („filename“);
```

- Unterverzeichnis:

```
File dir = new File („dirname“);
```

Methoden von `File`

Methoden für Dateien oder Unterverzeichnisse

- **String** `getName()` - liefert Datei- bzw. Verzeichnis-Namen
- `getAbsolutePath()` - liefert Datei-/Verzeichnis-Namen mit Pfad
- **String** `getParent()` - liefert das Oberverzeichnis
- **boolean** `exists()` - Existiert die Datei?
- **boolean** `canWrite()` - Darf in die Datei geschrieben werden?
- **boolean** `canRead()` - Darf die Datei ausgelesen werden?
- **boolean** `isFile()` - Ist es eine Datei, ...
- **boolean** `isDirectory()` ... oder ein Verzeichnis?
- **boolean** `mkdir()` - legt ein Verzeichnis an
- **String[]** `list()` - für eine Liste von allen in einem Verzeichnis enthaltenen Datei- und Verzeichnis-Namen.
- **boolean** `delete()` - für das Löschen des Files

(Für weitere Methoden von `File`, lesen Sie bitte in der Java API nach)

Bsp.: Klasse InfoTxt (Handling von Dateien)

```
import java.io.*;

public class InfoTxt {
    public static void main (String[] args){
        File info = new File("info.txt");
        //Erzeugt File-Objekt mit Namen "info.txt"
        System.out.println(info.getName());
        System.out.println(info.length());
    }
}
```

(vereinfachtes Beispiel – ohne Exception-Handling)

Bsp.: Klasse File

(Handling von Dateien)

```
import java.io.*;

public class FileTest{
    public static void main (String[] args){
        File file = new File("file.txt");
        //Erzeugt File-Objekt mit Dateiname „file.txt“
        if (file.exists() && !file.isDirectory()){
            System.out.println("Datei "+file.getName()+
                "gefunden");
            System.out.println("Voller Name: " +
                file.getAbsolutePath());
        }
        else if (!file.exists()) {
            System.out.println("Datei "+file.getName()+"
                existiert nicht!");
        }
    }
}
```

Bsp.: Klasse File – (Handling von Verzeichnissen)

```
import java.io.*;
public class DirTest{
    public static void main (String args[]){
        File dir=new File(".");//aktuelles Verzeichnis
        String[] list=dir.list();
        for (int i=0; i<list.length; i++)
            if (list[i].endsWith(".txt")){
                long length=new File(dir,list[i]).length();
                // Länge der Files
                System.out.println(list[i]+" ["+length+"]");
            }
    }
}
```

Datenströme (Streams)

IO-Operationen werden in Java durch Datenströme realisiert:

Ein **Datenstrom** (engl.: Stream) ist eine Datenstruktur, welche Daten in serieller Form speichern kann.

Mit einem Stream kann entweder

- **gelesen** oder
- **geschrieben** werden.

Reading und Writing

- Wenn ein Programm Daten lesen soll, dann wird ein Stream auf die Datenquelle geöffnet und die Information seriell gelesen.



- Ebenso kann ein Programm Daten schreiben. Ein Stream wird zu einer Datenquelle geöffnet und die Information seriell hinausschreiben.



Ablauf

- **Lesen**

1. Öffnen eines Streams;
2. Solange noch Info
3. Lese Info;
4. Schließe den Stream;

- **Schreiben**

1. Öffnen des Streams;
2. Solange noch Info
3. Schreibe Info;
4. Schließe den Stream;

**Selber Ablauf -
egal woher die Daten kommen und wohin sie gehen!**

Arten von Datenströmen

Es gibt in Java zwei verschiedene Gruppen von Datenströmen:

- **Byte-Streams** (`InputStream`, `OutputStream`) werden für das Arbeiten mit byte-orientierten Datenströmen eingesetzt. Die Transporteinheiten bei Byte-Streams sind 8 Bit lang.
- **Character-Streams** (`Reader` und `Writer`) werden für das Arbeiten mit zeichen- und zeilenorientierten Datenströmen eingesetzt. Diese Klassen verwenden 16 Bit lange Unicode-Zeichen.

Reader vs. InputStream

Writer vs. OutputStream

Reader vs. InputStream

- Reader
 - Lesen von Character-Streams (16 bit): Zeichen und Arrays von Zeichen
 - Textdateien
- InputStream
 - Lesen von Byte-Streams (8 Bit)
 - Binäre Daten wie Soundfiles, Bilder

Writer vs. OutputStream

- Writer
 - Schreibt Character-Streams
- OutputStream
 - Schreibt Byte-Streams

Klasse InputStream

InputStream ist die abstrakte **Oberklasse** für das **Lesen** von Datenströmen, also für die Byte-orientierte **Eingabe** (z.B. Bild- und Sounddateien).

`FileInputStream` ist eine Subklasse von `InputStream` für das Lesen von Dateien. Das Öffnen des Lese-Stroms bzw. des Files erfolgt mit einem Konstruktor der Form

```
FileInputStream infile =  
    new FileInputStream (fileObject);
```

(Weitere Subklassen sind z.B.

`ObjectInputStream` oder `ByteArrayInputStream`)

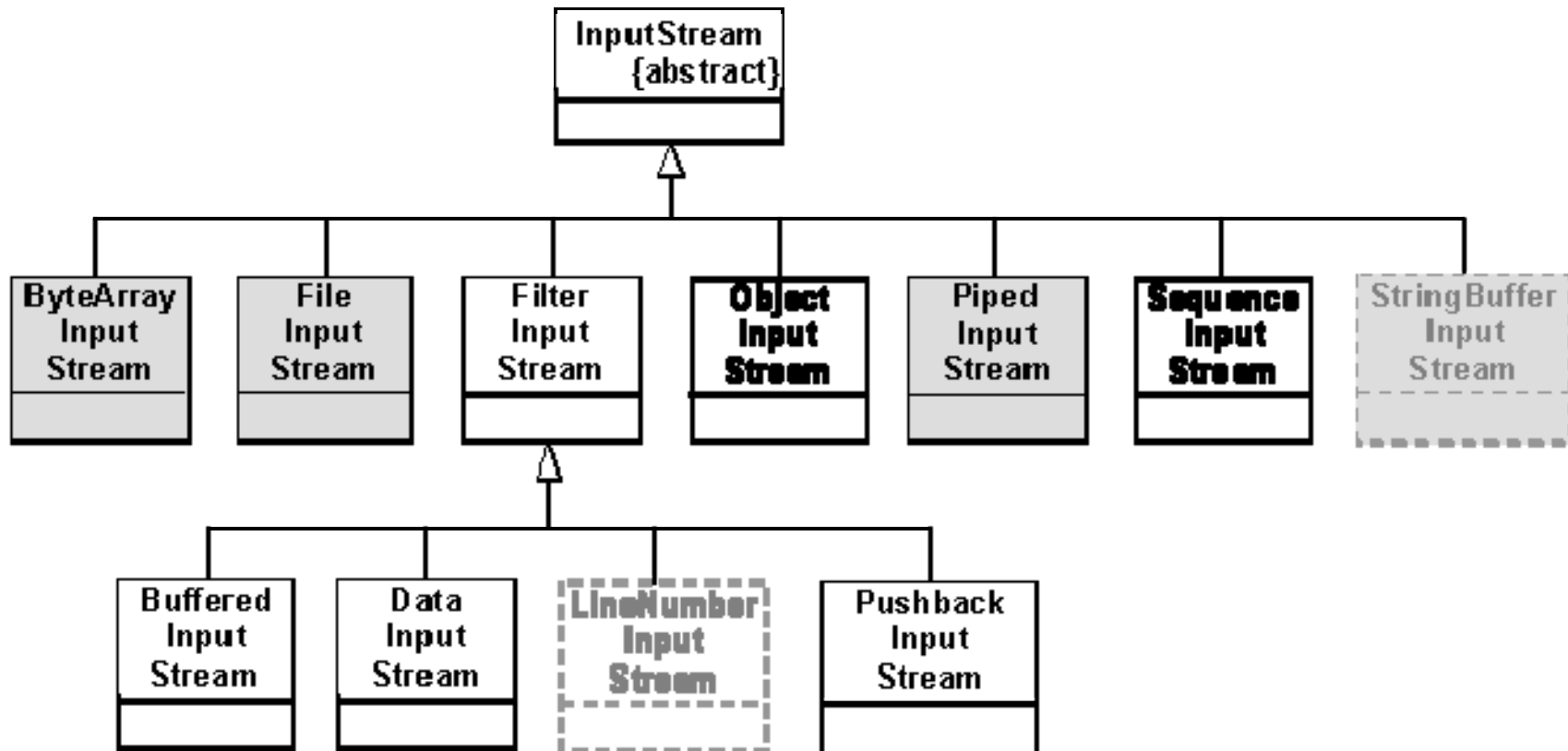
Klasse InputStream II

- Wichtige Methoden der Klasse
InputStream:
- **int** read()
liefert ein Byte, oder den Wert -1, wenn das Ende des Datenstroms (End-of-File) erreicht wurde.
- **void** close() Beendet eine Stream-Verbindung und gibt alle Systemressourcen frei.

InputStream

Klassenhierarchie

Die Klasse `InputStream` und ihre Subklassen:



Klasse OutputStream

`OutputStream` ist die abstrakte **Oberklasse** für das **Schreiben** von Datenströmen, also für die Byte-orientierte **Ausgabe**.

`FileOutputStream` ist eine Subklasse von `OutputStream` für das Schreiben von Dateien. Das Öffnen des Schreib-Stroms bzw. des Files erfolgt mit einem Konstruktor der Form

```
FileOutputStream outfile =  
    new FileOutputStream(fileObject);
```

(Weitere Subklassen sind z.B. `ByteArrayOutputStream`, oder `ObjectOutputStream`)

Klasse OutputStream II

Wichtige Methoden der Klasse OutputStream:

- **int** write()

liefert ein Byte, oder den Wert -1, wenn das Ende des Datenstroms (End-of-File) erreicht wurde.

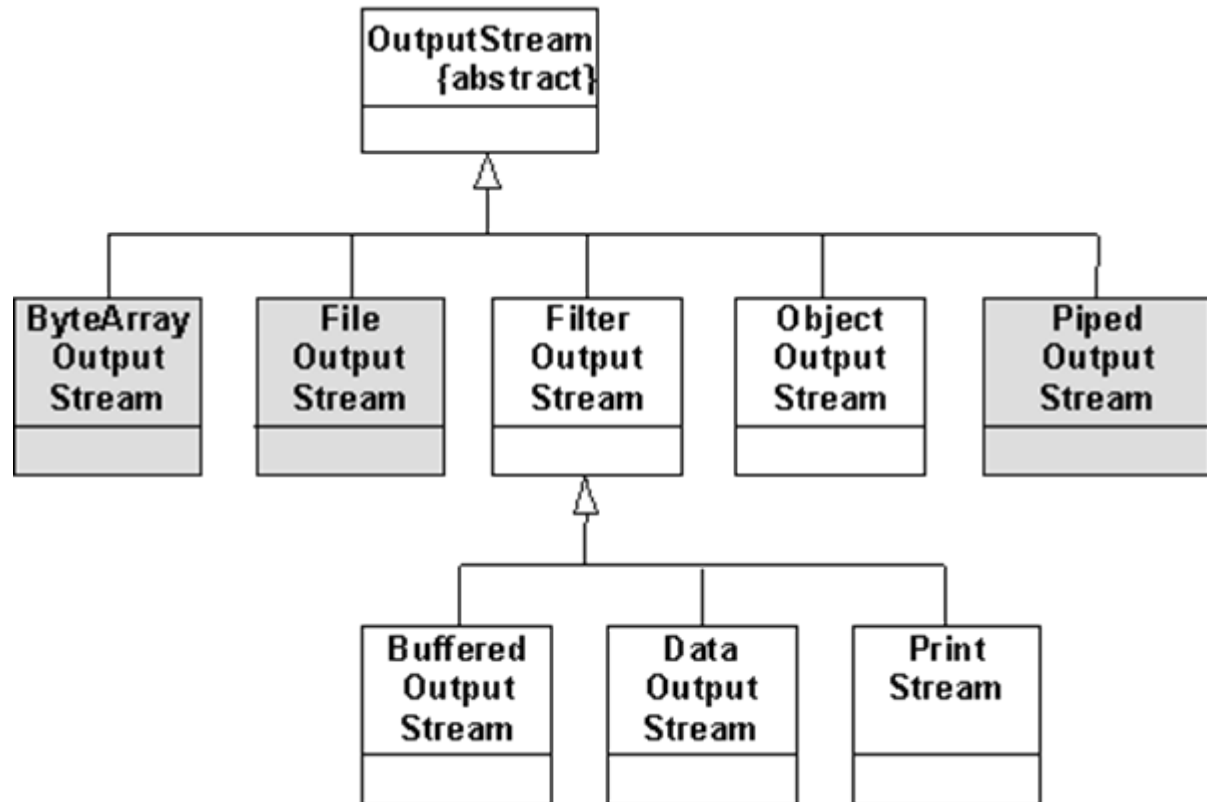
- **abstract void** close()

Beendet die Stream-Verbindung und gibt alle Systemressourcen frei.

OutputStream

Klassenhierarchie

Die Klasse OutputStream und ihre Subklassen:



Kopieren von Dateien

Beispiel

```
import java.io.*;

public class FileStream {
    public static void main(String[] args){

        File inputFile = new File("music.mp3");
        File outputFile = new File("song.mp3");
        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);

        int c;
        while ((c = in.read()) != -1){
            out.write(c);
        }
        in.close();
        out.close();
    }
}
```

(vereinfachtes Beispiel – ohne Exception-Handling. Vollst. Code: siehe Copy.java)

Klasse Reader

Reader ist die abstrakte **Oberklasse** für das **Lesen** von Textströmen, also für die Zeichen-orientierte **Eingabe**.

`InputStreamReader` ist eine Subklasse von `Reader` für das Lesen von Dateien. Das Öffnen des Lese-Stroms bzw. des Files erfolgt mit einem Konstruktor der Form

```
InputStreamReader infile =  
    new FileReader(fileObject);
```

(Weitere Subklassen sind z.B.

`StringReader` oder `CharArrayReader`)

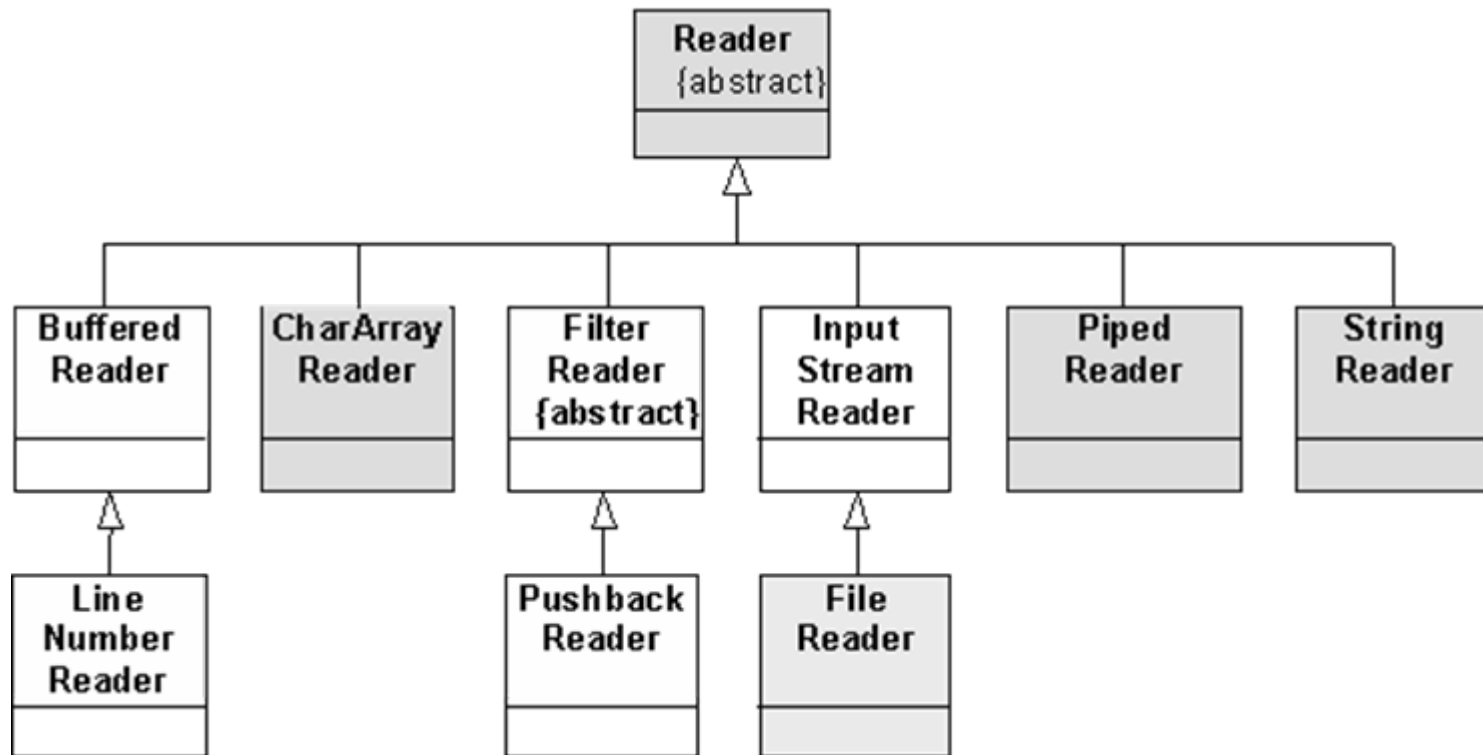
Klasse Reader II

- Wichtige Methoden der Klasse Reader:
- **int** read()
liefert ein Zeichen, oder den Wert -1, wenn das Ende des Datenstroms (End-of-File) erreicht wurde.
- **abstract void** close() Beendet eine Stream-Verbindung und gibt alle Systemressourcen frei.

Reader

Klassenhierarchie

Die Klasse `Reader` und ihre Subklassen:



Klasse `Writer`

`Writer` ist die abstrakte **Oberklasse** für das **Schreiben** von Textströmen, also für die Zeichen-orientierte **Ausgabe**.

`OutputStreamWriter` ist eine Subklasse von `Writer` für das Schreiben von Dateien. Das Öffnen des Schreib-Stroms bzw. des Files erfolgt mit einem Konstruktor der Form

```
OutputStreamWriter outfile =  
    new OutputStreamWriter(fileObject);
```

(Weitere Subklassen sind z.B. `PrintWriter` oder `StringWriter`)

Klasse `Writer` II

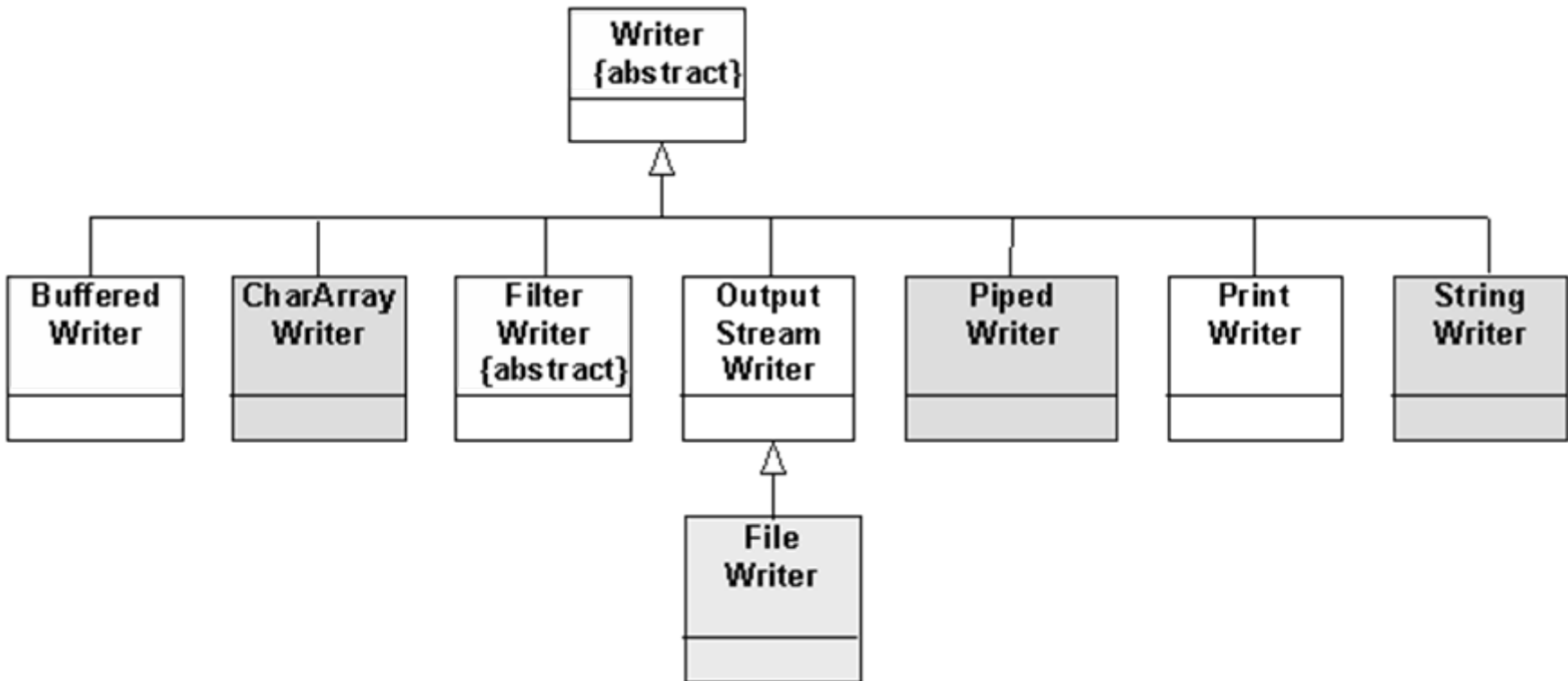
Wichtige Methoden der Klasse `Writer`:

- **`int write()`**
liefert ein Zeichen, oder den Wert `-1`, wenn das Ende des Datenstroms (End-of-File) erreicht wurde.
- **`abstract void close()`** Beendet eine Stream-Verbindung und gibt alle Systemressourcen frei.

Writer

Klassenhierarchie

Die Klasse `Writer` und ihre Subklassen:



Kopieren von Dateien

Beispiel

```
import java.io.*;

public class Copy {
    public static void main(String[] args){

        File inputFile = new File("input.txt");
        File outputFile = new File("outagain.txt");
        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);

        int c;
        while ((c = in.read()) != -1){
            out.write(c);
        }
        in.close();
        out.close();
    }
}
```

(vereinfachtes Beispiel – ohne Exception-Handling. Vollst. Code: siehe Copy.java)

Buffer

- zum Verbessern der Effizienz arbeitet man mit einem **Buffer**
- zum **Lesen**:
 - `BufferedReader`: **Character** bzw.
 - `BufferedInputStream`: **Byte**
- zum **Schreiben**:
 - `BufferedWriter`: **Character** bzw.
 - `BufferedOutputStream`: **Byte**

Buffer 2

- Ohne Buffer bewirkt jeder Aufruf von `read()`, dass ein einzelnes Byte aus dem File gelesen und in den Datentyp `Character` übersetzt und dann so zurückgeliefert wird.
- Um die Eingabe effizienter und schneller zu machen, soll nicht jedes Byte einzeln gelesen werden, sondern aus einem Pufferbereich.
- Die `Buffered`-Klassen bieten zusätzliche Methoden.(z.B. die Klasse `BufferedReader` - `readLine()`)

BufferedReader und BufferedWriter

- Konstruktoren

- `new BufferedReader (Reader)`
- `new BufferedWriter (Writer)`

- Beispiele

```
File fileI = new FileReader („FileIn.txt“);
```

```
BufferedReader in =  
    new BufferedReader(fileI);
```

```
File fileO = new FileWriter („FileOut.txt“);
```

```
BufferedWriter out =  
    new BufferedWriter(fileO);
```

- Kurzform

```
BufferedReader in = new BufferedReader (  
    new FileReader ("ReadFile.java"));
```

Lesen von Dateien

Beispiel

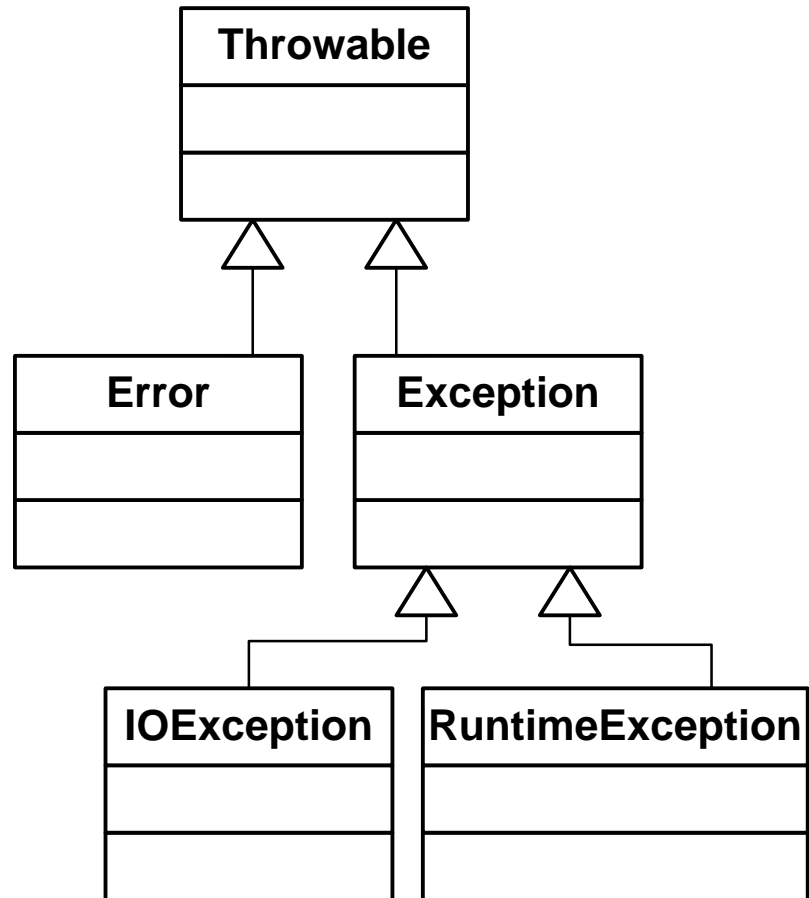
```
import java.io.*;
public class ReadFile {
    public static void main (String[] args) {
        String thisLine;
        File fileI = new FileReader („FileIn.txt“);
        BufferedReader in = new BufferedReader(fileI);

        while( (thisLine = in.readLine()) != null) {
            System.out.println(thisLine);
        }
        in.close();
    }
}
```

(vereinfachtes Beispiel – ohne Exception-Handling.)

Exceptions

- Alle Ein-/ und Ausgabe-Operationen können Fehlersituationen auslösen:
- Alle I/O-Operationen müssen durch Ausnahmeregelungen (Exceptions) bei evtl. Fehlersituationen abgefangen werden.



Exceptions

- Eine Exception ist ein Ereignis, das während der Programmausführung auftritt und den normalen Ablauf des Programms stört bzw. abbricht.
- Java bietet Fehlerbehandlungsroutinen (Exception Handling), um während des Programmablaufes auftretende Exceptions abzufangen.
- Diese auftretenden Probleme werden durch Objekte spezieller Exception-Classes repräsentiert, welche eine genaue Fehlerbeschreibung und den derzeitigen Zustand des Programmes speichern und ausgeben können.

Vorgänge beim Lesen eines Files

Pseudocode:

```
readFile {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```

Mögliche Exceptions

Was passiert wenn...

- die Datei nicht geöffnet werden kann?
- die Länge der Datei nicht bestimmt werden kann?
- nicht genügend Speicher zugeteilt werden kann?
- wenn das Lesen fehlschlägt?
- wenn das File nicht geschlossen werden kann?

Exception – Try/Catch Block Syntax

Der Try-Block umschließt die IO-Operationen. Für jeden auftretenden Fehler kann eine Fehlerbehandlungsroutine aus einer oder mehreren Anweisungen definiert werden (Catch-Block). Die „geworfenen“ (thrown) Exceptions werden über eine typisierte Catch - Anweisung abgefangen, sofern die Ausnahme mit ihrem Typ übereinstimmt.

```
import java.io.*
class ReadFile{
    try { open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (FileNotFoundException e){doSomething;}
    catch (NotOwnerException e) { doSomething; }
    catch (IllegalAccessException e) {doSomething;}
    ... }
```

Exception - Syntax

Alle Methoden in Java, die im Laufe ihrer Ausführung eine Exception auslösen könnten, können diese generell ankündigen. Dies geschieht bei der unmittelbaren Definition der Methode, und zwar durch das `throws` - Schlüsselwort im Kopf der Definition. Alternativ kann in der Methode `try/catch` verwendet werden. Die beiden Konzepte können auch kombiniert werden.

Die Methode gibt nur standardmäßige Fehlermeldungen aus:

```
type methodeName() throws Exception {  
// Anweisungen  
}
```

Lesen von Dateien

Klasse ReadFile hier mit Exception-Handling

```
import java.io.*;
public class ReadFile {
    public static void main (String[] args) {
        String thisLine;
        try {
            BufferedReader in = new BufferedReader (
                new FileReader ("ReadFile.java"));
            while( (thisLine = in.readLine()) != null) {
                System.out.println(thisLine);
            }
            in.close();
        } catch (Exception e) {
            System.out.println("error " + e);
        }
    }
}
```

Schreiben von Dateien

Beispiel

```
import java.io.*;
public class WriteFile {
    public static void main (String[] args) {
        try {
            BufferedWriter out =
                new BufferedWriter(
                    new FileWriter ("file.txt"));
            // FileWriter fw = new FileWriter("file.txt");
            // BufferedWriter out = new BufferedWriter(fw);
            out.write("Der Text für die Datei");
            out.newLine();
            out.close();
        }
        catch (Exception e) {System.out.println(e);}
    }
}
```

Kopieren von Dateien

Klasse Copy hier mit Exception-Handling

```
import java.io.*;

public class Copy {
    public static void main(String[] args) throws Exception {
        File inputFile = new File("input.txt");
        File outputFile = new File("outagain.txt");
        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
            in.close();
            out.close();
        }
    }
}
```

siehe Copy.java

Exception Beispiel

Wenn die Datei input.txt nicht im aktuellen Verzeichnis enthalten ist, wird beim Aufruf von der Klasse „Copy“ die FileNotFoundException ausgegeben. Diese Exception wird in der Klasse FileNotFoundException definiert und ist eine Subklasse von IOException.

```
Exception in thread "main" java.io.FileNotFoundException: input.txt (Das
System kann die angegebene Datei nicht finden)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(Unknown Source)
  at java.io.FileInputStream.<init>(Unknown Source)
  at java.io.FileReader.<init>(Unknown Source)
  at Copy.main(Copy.java:7)
```

Beispiel: Abfangen spezieller Exceptions (FileNotFoundException)

```
import java.io.*;

public class Copy {
    public static void main(String[] args) {
        try {
            File inputFile = new File("input.txt");
            FileReader in = new FileReader(inputFile);
            int c;
            while ((c = in.read()) != -1)
                System.out.print(c);
            in.close();
        } catch (FileNotFoundException fnf) {
            System.err.println("Sondermeldung: Input.txt not found! "+fnf);
        } catch (Exception e) {System.err.println(e);
        } }
    }
```

Erzeugt folgende Fehlermeldung, wenn es die Datei input.txt nicht gibt:
Sondermeldung: Input.txt not found! java.io.FileNotFoundException:
input.txt (Das System kann die angegebene Datei nicht finden)

Tastatureingabe

Klasse Echo mit Exception-Handling

```
import java.io.*;

public class Echo {
    public static void main(String[] args)
        throws IOException {
        BufferedReader into = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = into.readLine()).length() != 0)
            //solange nicht Leerzeile
            System.out.println(s);
            //eine Leerzeile beendet das Programm
        }
    }
}
```

Lernkontrolle

- Sie kennen die Syntax, für die Eingabe und Ausgabe von Daten eines Java-Programms.
- Sie kennen die Unterschiede und die Anwendung der Klassen `Reader`, `Writer`, `InputStream`, `OutputStream`
- Bei der Ein- und Ausgabe von Daten können Fehler auftreten. Sie wissen, wie Sie diese Fehler abfangen können.

