

# Programmiertechnik 1

## Teil 6: Ein-/Ausgabe


Streams, Dateizugriff, Objekt-Serialisierung



# Java Ein-/Ausgabe: Streams

---

Die Java Standard-Bibliothek definiert im Paket `java.io` **Streams** als Abstraktionen für Eingabe-Quellen und Ausgabe-Ziele:

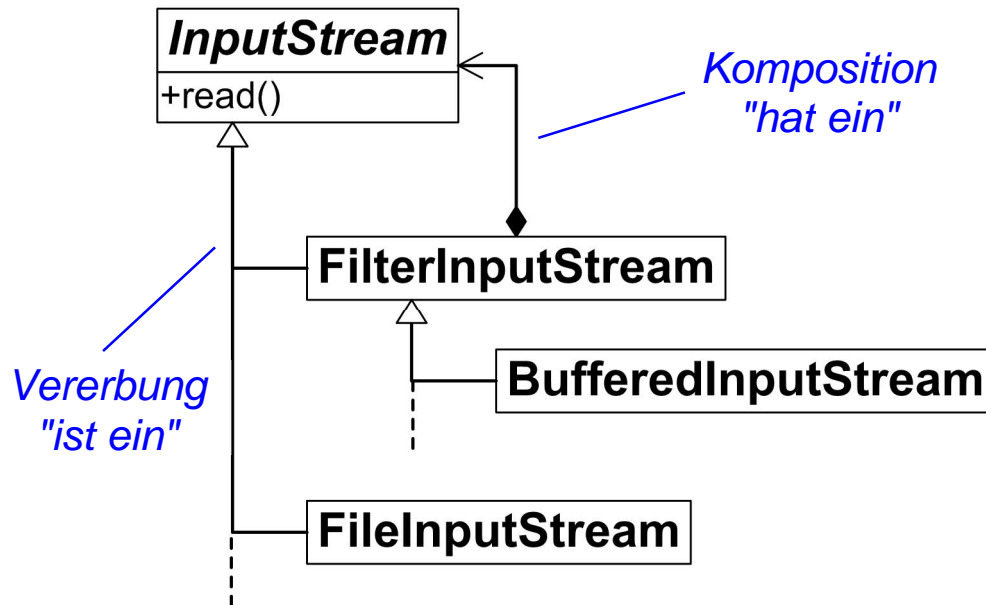
- Eingabe-Quellen sind z.B.:   
Standardeingabe (*Tastatur*), Dateien (*Hintergrundspeicher*),  
Felder / Strings (*Hauptspeicher*), ...
- Ausgabe-Ziele sind z.B.:  
Standardausgabe (*Bildschirm*), Dateien (*Hintergrundspeicher*),  
Felder / Strings (*Hauptspeicher*), ...

Es gibt zwei Arten von Streams:

- **Byte-Streams** zum Lesen und Schreiben von Bytes  
*für die Ein-/Ausgabe binärer Daten gedacht, deshalb für Text nur eingeschränkt nutzbar*
- **Character-Streams** zum Lesen und Schreiben von Zeichen  
*für die Ein-/Ausgabe von Text gedacht, verarbeiten UTF-16-Zeichen und unterstützen die Abbildung auf andere Zeichensätze*

# Java Byte-Streams: Klasse `java.io.InputStream`

Die Klasse `java.io.InputStream` ist Oberklasse aller Eingabe-Byte-Streams:



- zentrale Instanzmethode:  
`public abstract`  
`int read()` throws `IOException`

*liefert das nächste zu lesende Byte  
als Zahl zwischen 0 und 255  
oder bei Eingabeende -1*



`System.in` ist eine polymorphe Klassenvariable ist vom Typ `InputStream`:

```
public static final InputStream in =  
    new BufferedInputStream(new FileInputStream(FileDescriptor.in)); 
```

*Für das Lesen von Zeichen muss `System.in` mit einem `java.util.Scanner`- oder einem `Character-Stream`-Objekt kombiniert werden, das Bytes auf UTF16-Codes abbildet.*

# Beispiel-Programm InputStream

```
import java.io.*;

public final class CountBytes {
    private CountBytes() { }
    public static void main(String[] args) throws IOException {
        InputStream in; 
        if (args.length == 0) {
            in = System.in;
        } else {
            in = new FileInputStream(args[0]); 
        }
        int total = 0;
        while (in.read() != -1) {
            ++total;
        }
        System.out.printf("%d Bytes\n", total);
    }
}
```

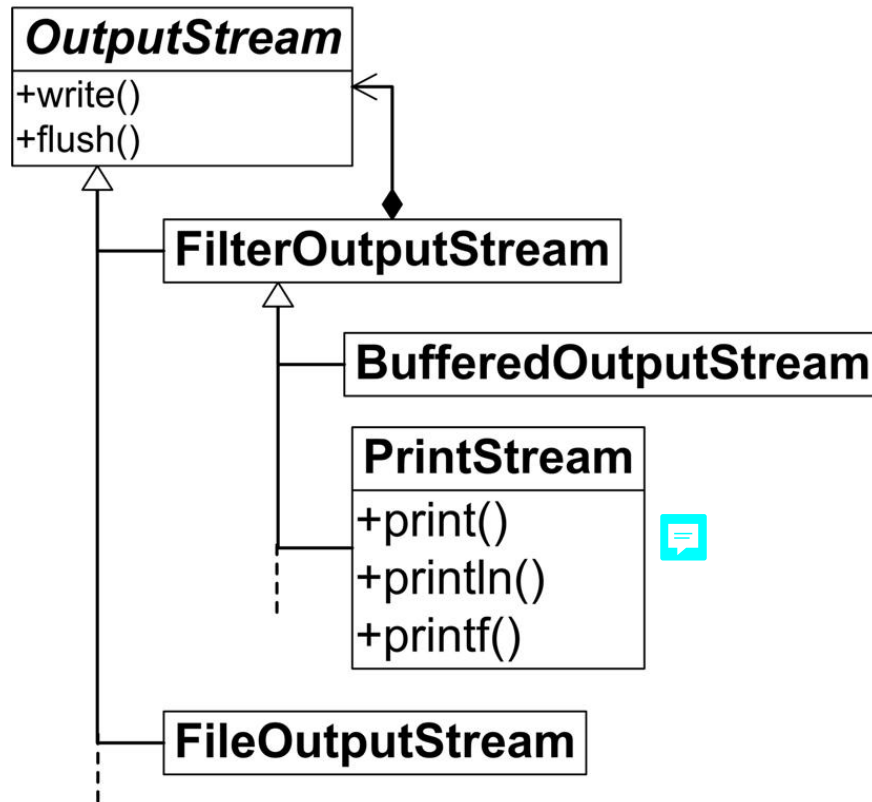
*Gibt die Anzahl der  
gelesenen Bytes aus*


*Deklaration erforderlich,  
weil IOException eine  
geprüfte Ausnahme ist*



# Java Byte-Streams: Klasse `java.io.OutputStream`

Die Klasse `java.io.OutputStream` ist Oberklasse aller Ausgabe-Byte-Streams:

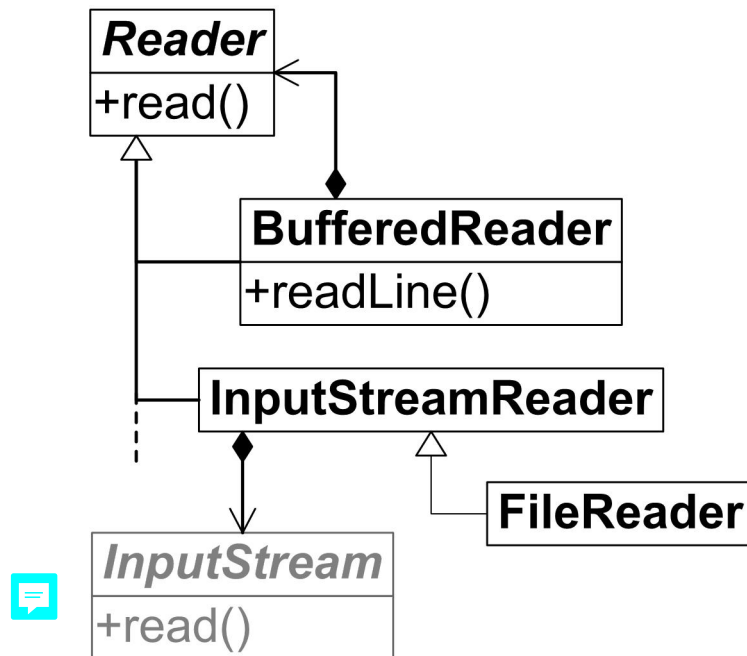


- Instanzmethoden:  
`public abstract`  
`void write(int b)` throws IOException  
*schreibt das niederwertigste Byte der übergebenen Zahl und ignoriert die übrigen 3 Bytes*  
`public`  
`void flush()` throws IOException  
*gibt eventuell gepufferte Bytes aus*  
... 

`System.out` / `System.err` sind polymorphe Klassenvariablen vom Typ `PrintStream`

# Java Character-Streams: Klasse `java.io.Reader`

Die Klasse `java.io.Reader` ist Oberklasse aller Eingabe-Character-Streams:




- zentrale Instanzmethode:

```
public abstract
int read(char[] buf, int offset, int count)
throws IOException
```

*liefert die nächsten maximal count Zeichen  
im Feld buf unter Index offset folgende und  
gibt die Anzahl gelesener Zeichen zurück  
oder bei Eingabeende -1*

Kombination aus Byte- und Character-Stream für das Lesen von Textzeilen:

```
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String s = in.readLine(); // liefert null bei Eingabeende
...
```

# Beispiel-Programm Reader

*Gibt die Anzahl der  
gelesenen Zeilen aus*

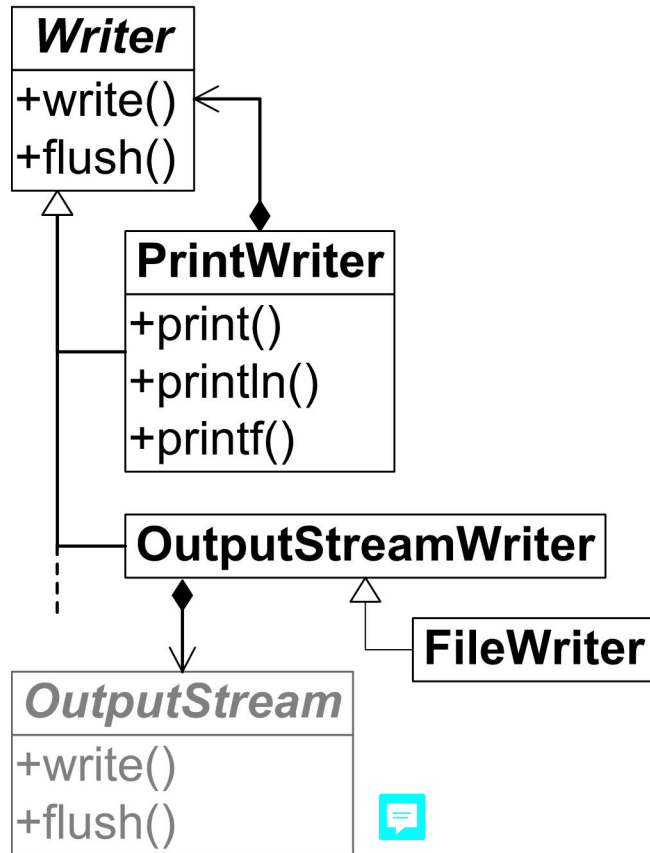
```
import java.io.*;

public final class CountLines {
    private CountLines() { }

    public static void main(String[] args) throws IOException {
        BufferedReader in;
        if (args.length == 0) {
            in = new BufferedReader(new InputStreamReader(System.in));
        } else {
            in = new BufferedReader(new FileReader(args[0]));
        }
        int total = 0;
        while (in.readLine() != null) {
            ++total;
        }
        System.out.printf("%d Lines\n", total);
    }
}
```

# Java Character-Streams: Klasse `java.io.Writer`

Die Klasse `java.io.Writer` ist Oberklasse aller Ausgabe-Character-Streams:



- zentrale Instanzmethoden:

`public abstract`

`void write(char[] buf, int offset, int count)`

`throws IOException`

*schreibt count Zeichen aus buf  
beginnend beim Index offset*

`public abstract`

`void flush() throws IOException`

*gibt eventuell gepufferte Zeichen aus*

...



# Java Character-Streams: Paket `java.nio.charset`

---

Das Paket `java.nio.charset` enthält Klassen für den Umgang mit **Character-Sets** zur Abbildung von Unicode-Zeichen auf Byte-Folgen und umgekehrt.

- die Klassen des Pakets werden normalerweise nicht explizit verwendet, sondern implizit über `OutputStreamWriter` bzw. `InputStreamReader`, z.B.:

```
new OutputStreamWriter( System.out, "UTF-8" );  
new InputStreamReader( System.in, "UTF-8" );
```

*beim Konstruktoraufruf kann ein Character-Set-Name übergeben werden (hier: "UTF-8")*

*beim Konstruktoraufruf ohne Character-Set-Name verwenden die Stream-Klassen eine plattformspezifische Standardeinstellung: `Charset.defaultCharset().name()`*

- von jeder Java-Implementierung unterstützte Character-Sets:

US-ASCII  
ISO-8859-1  
UTF-8  
UTF-16, UTF-16LE, UTF-16BE

*Abfrage aller unterstützten Character-Sets:  
`Charset.availableCharsets().keySet()`*

# Beispiel-Programm Charset

---

```
import java.io.*;
import java.nio.charset.Charset;

public final class FileEncoding {
    private FileEncoding() { }

    public static void main(String[] args) throws IOException {
        String charsetName = Charset.defaultCharset().name();
        if (args.length > 0) {
            charsetName = args[0];
        }

        String umlaute = "\u00E4\u00F6\u00FC\u00DF"; // ä ö ü ß
        String euro = "\u20AC"; // €
        String symbole = "\u00BD\u00B2\u221A\u2211"; // ½²√Σ
        String fileName = "charset-" + charsetName + ".txt";
        PrintWriter pw = new PrintWriter(fileName, charsetName);
        pw.println(umlaute + euro + symbole);
        pw.close();
    }
}
```

*Konstruktor erzeugt intern  
ein OutputStreamWriter-Objekt*

# Java Ein-/Ausgabe: Dateien (1)

---

Dateien sind benannte Bereiche in externem Speicher (*Festplatte, USB-Stick, ...*).

- **Gewöhnliche Dateien** haben einen beliebigen anwendungsspezifischen Inhalt, auf den über Byte-, Character-, Data- oder Object-Streams zugegriffen wird
- **Verzeichnisse** (*Ordner, Directories, Folder*) sind spezielle Dateien, die die Namen anderer Dateien enthalten
- seit Java 7 gibt es das **Paket `java.nio.file`** mit gegenüber älteren Versionen verbesserten Schnittstellen und Klassen für den Umgang mit Dateien:

Utility-Klasse **`java.nio.file.Paths`** und Schnittstelle **`java.nio.file.Path`** 

> *für den Umgang mit Dateinamen*

Utility-Klasse **`java.nio.file.Files`**

> *für das Anlegen, Löschen, Kopieren und Umbenennen von Dateien*

> *für den Umgang mit Dateieigenschaften (z.B. Existenz, Art der Datei, Zugriffsrechte, ...)*

diverse weitere Klassen und Schnittstellen ...

## Java Ein-/Ausgabe: Dateien (2)

---

Dateien sollten geschlossen werden, sobald kein Zugriff mehr gebraucht wird

> *geschriebene Daten können sonst in Hauptspeicherpuffern "hängen bleiben"*

> *offene Dateien belegen beschränkte Ressourcen im Betriebssystem*

- die Methode zum Schließen von Datei-Streams ist in den Schnittstellen java.io.Closeable und java.lang.AutoCloseable definiert (*letztere ab Java 7*):

```
void close() throws IOException
```

*alle Stream-Klassen implementieren die Schnittstellen*

- im Zusammenhang mit auftretenden Ausnahmen war der korrekte Umgang mit close( )-Aufrufen in Java traditionell etwas schwierig, ist aber mit der Syntax try-with-resources (*ab Java 7*) deutlich einfacher geworden:


```
try (  
    RessourcenKlasse r = new RessourcenKlasse( ... );  
    ) {  
    ... // Ressourcenzugriff  
    } // automatischer Aufruf r.close( ) auch bei Ausnahmen
```

*RessourcenKlasse  
muss AutoCloseable  
implementieren  
mehrere Ressourcen  
sind möglich*

# Beispiel-Programm Dateien


```
import java.io.*;
import java.nio.file.*;

public final class CopyFile {
    private CopyFile() { }

    public static void main(String[] args) throws IOException {
        try (
             InputStream in = Files.newInputStream(Paths.get(args[0]));
            OutputStream out = Files.newOutputStream(Paths.get(args[1]));
        ) {
            int b;
            while ((b = in.read()) != -1) {
                out.write(b);
            }
        }
    }
}
```

*Kopiert eine Datei  
(sehr vereinfacht ohne  
Fehlerbehandlung)*

*automatische Aufrufe `out.close()` und `in.close()`*

 *Einfacher ist das Kopieren mit `Files.copy`:*  
`Files.copy(Paths.get(args[0]), Paths.get(args[1]));`

# Beispiel-Programm Verzeichnisse

*Listet Verzeichnisse und  
ihre Unterverzeichnisse auf*

```
import java.io.IOException;
import java.nio.file.*;

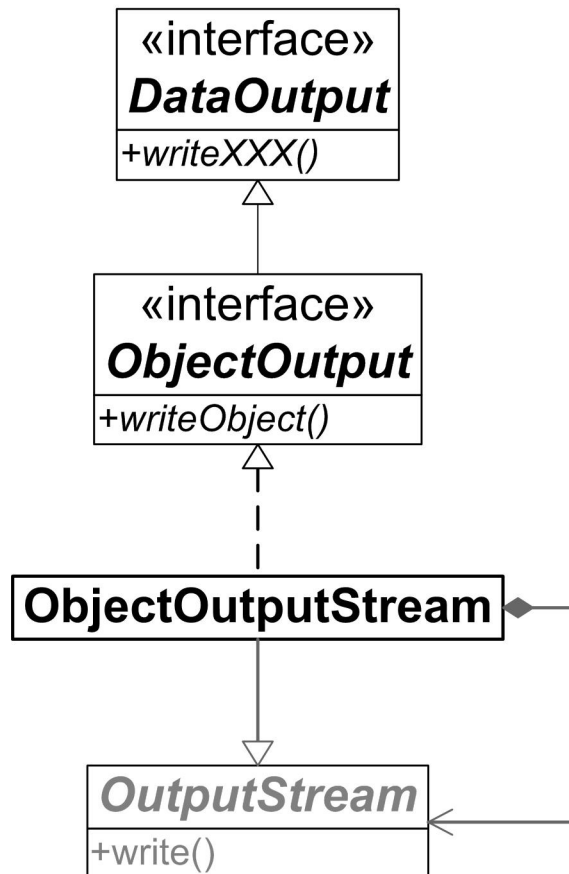
public final class ListFiles {
    private ListFiles() { }
    public static void main(String[] args) throws IOException {
        for (String s : args) {
            Path p = Paths.get(s);
            if (Files.exists(p)) { list(p); }
        }
    }
    private static void list(Path p) throws IOException {
        System.out.println(p);
        if (Files.isDirectory(p)) {
            try ( DirectoryStream<Path> d = Files.newDirectoryStream(p); ) {
                for (Path entry : d) { list(entry); }
            }
        }
    }
}
```

*rekursiverer Aufruf ListFiles.list( Path)*

*automatischer Aufruf d.close( )*

# Java Ein-/Ausgabe: Objekt-Serialisierung (1)

Die Klasse [java.io.ObjectOutputStream](#) serialisiert Java-Daten, d.h. sie wandelt Java-Daten in Bytefolgen: 



- zentrale Instanzmethoden:

**public**

**void writeXXX(**XXX** v)** throws IOException

*schreibt einen Wert des Typs XXX als Bytefolge,  
z.B. writeInt(int v)*

**public**

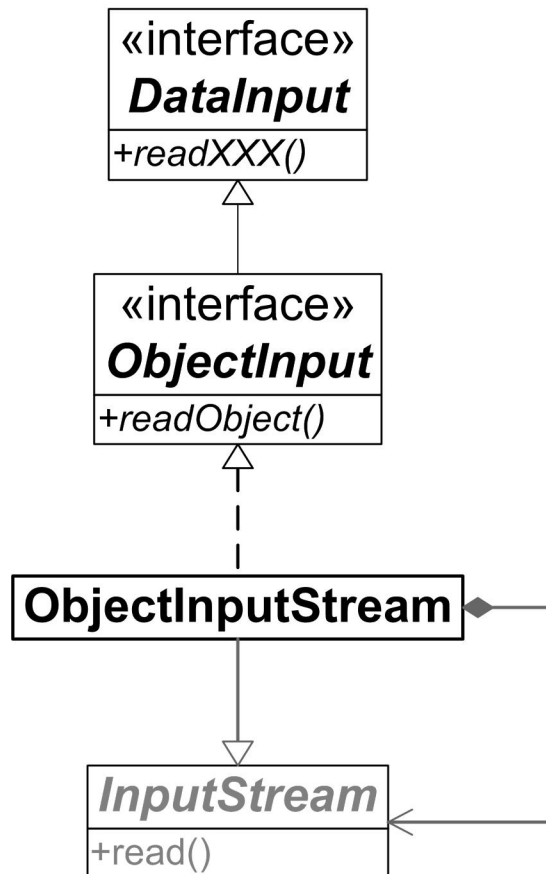
**void writeObject(Object obj)** throws IOException


*schreibt ein Objekt mit seinen Instanzvariablen  
als Bytefolge, inklusive der referenzierten Objekte  
die Klassen der Objekte müssen dazu  
mit dem Interface java.io.Serializable markiert sein*



# Java Ein-/Ausgabe: Objekt-Serialisierung (2)

Die Klasse [java.io.ObjectInputStream](#) deserialisiert Java-Daten, d.h. rekonstruiert Java-Daten aus Bytefolgen:



- zentrale Instanzmethoden:  
`public XXX readXXX() throws IOException`  
*rekonstruiert einen Wert des Typs XXX aus einer Bytefolge, z.B. `int readInt()`*  
  
`public Object readObject() throws ClassNotFoundException, IOException`  
*rekonstruiert ein Objekt mit seinen Instanzvariablen aus einer Bytefolge* 





# Beispiel-Programm Objekt-Serialisierung


*Serialisiert und deserialisiert  
ein Objekt*

```
import java.io.*;
import java.nio.file.*;

public final class SerializeObject {
    private SerializeObject() { }

    private static final class Beispiel implements Serializable { }

    public static void main(String[] args) throws Exception {
        Path p = Paths.get("beispiel.ser"); 
        Beispiel b = new Beispiel();
        ObjectOutput out = new ObjectOutputStream(Files.newOutputStream(p));
         out.writeObject(b);
        out.close();

        ObjectInput in = new ObjectInputStream(Files.newInputStream(p));
         Beispiel bb = (Beispiel) in.readObject();
        in.close();

        System.out.printf("%s%n%s%n", b, bb);
    }
}
```

# Java Ein-/Ausgabe: Index

---

`.close()` 6-11

Byte-Stream 6-1 bis 6-4

Character-Set 6-8,6-9

Character-Stream 6-1,6-5 bis 6-7

IOException 6-2,6-4,6-5,6-7,6-14,6-15

`java.io.InputStream` 6-2,6-3

`java.io.OutputStream` 6-4

`java.io.Closeable` 6-11

`java.io.PrintStream` 6-4

`java.io.Reader` 6-5,6-6

`java.io.Writer` 6-7

`java.lang.AutoCloseable` 6-11

`java.nio.file.Files` 6-10

`java.nio.file.Path` 6-10

`java.nio.file.Paths` 6-10

Objekt-Serialisierung 6-14 bis 6-16

Stream 6-1

`System.err` 6-4

`System.in` 6-2

`System.out` 6-4

`try-with-resources` 6-11