Ü 1.1 Java-I/O Einführung

a) Klassen OutputStream und InputStream

Die Klassen **OutputStream** und **InputStream** sind die Basisklassen für die Byte-orientierte Ein- und Ausgabe.

- **InputStream**:

Diese abstrakte Klasse repräsentiert den Eingabestrom von Bytes. Sie wird verwendet, um Daten Byte für Byte von einer Quelle zu lesen.

Wichtige Methoden:

- int read() liest ein Byte und gibt dessen Wert zurück (oder -1, wenn das Ende erreicht ist).
- int read(byte[] b) liest mehrere Bytes in ein Array und gibt die Anzahl gelesener Bytes zurück.
- Beispiel:

```
InputStream input = new FileInputStream("file.txt");
int data = input.read(); // liest das erste Byte aus der Datei
while(data != -1) {
    System.out.print((char) data);
    data = input.read();
}
input.close();
```

- **OutputStream**:

Diese abstrakte Klasse repräsentiert den Ausgabestrom von Bytes. Sie wird verwendet, um Daten Byte für Byte an eine Zielressource zu schreiben.

Wichtige Methoden:

- void write(int b) schreibt ein Byte in den Ausgabestrom.
- void write(byte[] b) schreibt ein Byte-Array in den Ausgabestrom.
- Beispiel:

```
OutputStream output = new FileOutputStream("output.txt");
output.write(65); // schreibt das Byte 65 ('A') in die Datei
output.close();
```

b) FilterInputStreams und FilterOutputStreams

FilterInputStream und **FilterOutputStream** sind abstrakte Klassen, die dekorierte Ein- und Ausgabeströme darstellen.

Sie bieten zusätzliche Funktionalität wie Pufferung oder Komprimierung, indem sie bestehende Streams erweitern.

c) Erklärungen der Klassen

1. **BufferedInputStream**

BufferedInputStream fügt Pufferung zur Datenlesung hinzu, um die Effizienz zu steigern. Statt jedes Byte einzeln zu lesen,

werden größere Blöcke im Speicher gepuffert.

- Beispiel:

```
InputStream input = new BufferedInputStream(new FileInputStream("file.txt"));
int data = input.read();  // liest Daten effizient durch Pufferung
input.close();
```

2. **DataOutputStream**

DataOutputStream ermöglicht es, Java-Datentypen (z.B. int, double) direkt in einen Ausgabestrom zu schreiben.

- Beispiel:

```
DataOutputStream output = new DataOutputStream(new FileOutputStream("data.bin")
output.writeInt(42);   // schreibt einen int-Wert als Byte-Array
output.writeDouble(3.14); // schreibt einen double-Wert
output.close();
```

3. **CipherInputStream**

CipherInputStream ermöglicht das Lesen von verschlüsselten Daten und entschlüsselt diese automatisch während des Lesens.

- Beispiel:

```
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.DECRYPT_MODE, secretKey);
InputStream input = new CipherInputStream(new FileInputStream("encrypted.dat"),
int data = input.read();  // liest und entschlüsselt die Daten
input.close();
```

4. **ZipOutputStream**

ZipOutputStream wird verwendet, um Daten in das ZIP-Format zu komprimieren und komprimierte Dateien in ZIP-Container zu schreiben.

- Beispiel:

```
ZipOutputStream zipOut = new ZipOutputStream(new FileOutputStream("archive.zip"));
zipOut.putNextEntry(new ZipEntry("file.txt"));
zipOut.write("Hello, World!".getBytes());
zipOut.closeEntry();
zipOut.close();
```

5. **PushbackInputStream**

PushbackInputStream erlaubt es, gelesene Bytes zurück in den Stream zu schieben, wenn man vorab gelesene Daten erneut interpretieren möchte.

- Beispiel:

```java
public static void main(String[] args) {
    try {
        // Datei öffnen
        FileInputStream fileStream = new FileInputStream("file.txt");

        // PushbackInputStream initialisieren
        PushbackInputStream pushbackStream = new PushbackInputStream(fileStream);

        // Erstes Zeichen lesen
        int data = pushbackStream.read();

        // Überprüfen, ob das Zeichen '<' ist
        if (data == '<') {
            System.out.println("Das Zeichen '<' wurde gefunden, es wird zurückgeschoben.");

            // Das Zeichen zurück in den Stream schieben
            pushbackStream.unread(data);

            // Nochmals lesen (dieses Mal das zurückgeschobene Zeichen)
            data = pushbackStream.read();
            System.out.println("Zeichen nach dem Zurückschieben erneut gelesen: " + (char) data);
        } else {
            System.out.println("Gelesenes Zeichen: " + (char) data);
        }

        // Stream schließen
        pushbackStream.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

d) Readers und Writers

Im Gegensatz zu den byte-orientierten Klassen (`InputStream`, `OutputStream`) arbeiten **Readers** und **Writers** zeichenorientiert. Sie sind darauf ausgelegt, mit **Unicode-Zeichen** anstelle von Bytes zu arbeiten und sind damit ideal für den Umgang mit Textdaten.

Reader:

- Diese abstrakte Klasse ist die Basisklasse für zeichenorientierte Eingabeströme.
- Sie bietet Methoden zum Lesen von Zeichen oder Arrays von Zeichen aus einer Textquelle (z.B. Datei, Konsole).
    - Beispiel:

```
Reader reader = new FileReader("file.txt");
int data = reader.read();  // liest ein Zeichen
reader.close();
```

BufferedReader ist eine Unterklasse von Reader und dient dazu, den Lesevorgang zu puffern, um die Effizienz zu steigern, insbesondere wenn viele kleine Lesevorgänge durchgeführt werden. Er gehört also zur **zeichenorientierten Verarbeitung** und sollte im Abschnitt zu **Readers und Writers** behandelt werden.

Writer:

- Diese abstrakte Klasse ist die Basisklasse für zeichenorientierte Ausgabeströme.
- Sie bietet Methoden zum Schreiben von Zeichen oder Arrays von Zeichen in eine Textquelle (z.B. Datei).
    - Beispiel:

```
Writer writer = new FileWriter("output.txt");
writer.write("Hello, World!");  // schreibt eine Zeichenkette in die Datei
writer.close();
```

Der **BufferedWriter** puffert die Ausgabedaten und schreibt sie erst in größeren Blöcken in die Ausgabesysteme, was die Leistung verbessert. Er wird häufig zusammen mit anderen Writer-Klassen verwendet, um effizienter Text zu schreiben.

Zusatz: BufferedWriter

BufferedWriter ist das Gegenstück zu **BufferedReader** und puffert Schreibvorgänge. Er hilft dabei, die Anzahl der physischen Schreiboperationen zu reduzieren, indem er Daten zunächst im Speicher sammelt und sie dann gebündelt schreibt.

Ü 1.5 Java Thread Safety

Thread-Safety bedeutet, dass eine Klasse oder Methode von mehreren Threads gleichzeitig ausgeführt werden kann, ohne dass unerwartete Ergebnisse oder Fehler auftreten. In Java ist dies besonders relevant, weil mehrere Threads oft gleichzeitig auf gemeinsame Ressourcen zugreifen, was zu **Race Conditions**, **Inkonsistenzen** oder **Deadlocks** führen kann.

Probleme bei fehlender Thread-Sicherheit:

- **Race Condition**: Tritt auf, wenn mehrere Threads gleichzeitig eine gemeinsam genutzte Variable ändern, was zu unerwarteten Ergebnissen führt.
- **Visibility-Probleme**: Änderungen eines Threads sind für andere Threads möglicherweise nicht sofort sichtbar.
- **Atomicität**: Nicht-atomare Operationen wie count++ können von mehreren Threads gleichzeitig unterbrochen und fehlerhaft ausgeführt werden.

Mechanismen zur Thread-Sicherheit:

- **Synchronisierung** (synchronized): Sorgt dafür, dass nur ein Thread gleichzeitig auf kritische Abschnitte zugreifen kann.

```
public synchronized void increment() { count++; }
```

- **volatile-Schlüsselwort**: Stellt sicher, dass eine Variable immer direkt aus dem Hauptspeicher gelesen wird und nicht zwischengespeichert wird.

```
private volatile boolean running = true;
```

- **Locks** (ReentrantLock): Bietet flexiblere Sperrmechanismen als synchronized.

```
lock.lock();
try { count++; } finally { lock.unlock(); }
```

- **Atomic Classes** (AtomicInteger, etc.): Bieten atomare Operationen auf Variablen.

```
AtomicInteger count = new AtomicInteger(0);
count.incrementAndGet();
```

Rechnernetze und Netzwerkprogrammierung

Aufgabenblatt 1 (Gruppe: Walcher, Dworski, Aigensberger)

Thread-sichere Klassen in Java:

- **Synchronized Klassen**: Vector, Hashtable, StringBuffer sind von Haus aus synchronisiert.
- **Concurrent Collections**: ConcurrentHashMap, CopyOnWriteArrayList und andere in java.util.concurrent sind speziell für Multi-Threading optimiert.

Best Practices:

- **Minimiere gemeinsamen Zustand**: Reduziert die Notwendigkeit von Synchronisation.
- **Immutable Objects**: Unveränderliche Objekte sind von Natur aus thread-sicher.

Fazit: Thread-Safety ist wichtig, um sicherzustellen, dass Programme korrekt und effizient in einer Multi-Thread-Umgebung funktionieren. Es gibt verschiedene Techniken und Klassen in Java, um dies zu gewährleisten, je nach Anwendungsfall.