universität innsbruck

Streams

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Interaktion mit dem Dateisystem

Ein- und Ausgabe

- Für die Ein- und Ausgabe stellt die Java-API zwei Pakte `java.io` und `java.nio` bereit.
  - Das Paket `java.io` gibt es seit JDK 1.0
  - Das Paket `java.nio` wurde mit Java 1.4 eingeführt.
  - Mit Java 7 würde die Ein- und Ausgabefunktionalität erweitert. Diese Erweiterung wird als NIO2 bezeichnet.
- Auszug wichtiger Elemente für den Zugriff auf das Dateisystem mit java.nio.file:
  - Interface `Path` – Repräsentiert einen Pfad im Dateisystem
  - Klasse `Files` – Hilfsklasse, welche ausschließlich statische Methoden für das Arbeiten mit Files zur Verfügung stellt.

Interface Path (1)

- Ein Exemplar, welche dieses Interface erfüllt, repräsentiert ein File in einem hierarchischen Pfad eines Dateisystems.
  - Ein File kann eine Datei oder ein Verzeichnis sein.
  - Die Datei oder das Verzeichnis, welches der Pfad referenziert, muss nicht existieren.
- Ein Pfad kann auf zwei Arten spezifiziert werden.
  - Absoluter Pfadname (von der Wurzel ausgehend)
  - Relativer Pfadname (abhängig vom aktuellen Verzeichnis)
- Pfadangaben sind plattformabhängig!
  - `C:\Users\Max\Documents\my_file.txt`
  - `/home/max/Documents/my_file.txt`

Interface Path (2)

- Auszug einiger Methoden:
  - `getFileName()`
    - Ermittelt den Dateiname bzw. Verzeichnisname
  - `getParent()`
    - Ermittelt den Pfad des übergeordneten Verzeichnis
  - `normalize()`
    - Normalisiert den Pfad
  - `of(String first, String... more)`
    - Statische Methode zum Erstellen eines Pfads
  - `toAbsolutePath()`
    - Liefert den absoluten Pfad

Klasse Files

- Die Hilfsklasse Files bietet ausschließlich statische Methoden an.
- Auszug einiger Methoden:

`createDirectory(Path dir, FileAttribute<?>... attrs)`
  - Erstellt ein neues Verzeichnis

`createFile(Path path, FileAttribute<?>... attrs)`
  - Erstellt eine neue, leere Datei

`exists(Path path, LinkOption... options)`
  - Überprüft ob das File existiert

`isDirectory(Path path, LinkOption... options)`
  - Überprüft of das File ein Verzeichnis ist

`list(Path dir)`
  - Ermittelt alle Elemente des Verzeichnisses als `java.util.stream.Stream`

`newBufferedReader(Path path)`
  - Öffnet eine Datei zum Lesen mit einen `BufferedReader`

`newBufferedWriter(Path path, OpenOption... options)`
  - Öffnet eine Datei zum Schreiben mit einen `BufferedWriter`

src/at/ac/uibk/pm/streams/FileApplication.java

Datenströme

Motivation

- Die Java Bibliothek enthält eine ganze Reihe von Klassen, um anspruchsvolle Ein- bzw. Ausgabeoperationen zu bewältigen und von verschiedenen Datenquellen zu lesen und zu schreiben.
- Das Ein- /Ausgabemodell in Java basiert auf sequentiellen Datenströmen.
  - Beim Schreiben erzeugt man einen Ausgabestrom (in eine Datei oder einen Speicher).
  - Beim Lesen kommen die Daten von einem Eingabestrom (von einer Datei oder einem Speicher).

I/O-Begriffe

- Datenstrom
  - Die interne Repräsentation einer (externen) Datei oder einer Input/Output-Einheit in einem System.
  - Sequentielle Abarbeitung
- Source/Destination
  - Jeder Stream hat eine Quelle (Source) und eine Senke (Sink, Destination).
- Input-Stream
  - Lesen der Daten von einer Datei oder einer Eingabeeinheit in einen Stream.
- Output-Stream
  - Schreiben der Daten von einem Stream in eine Datei oder eine Ausgabeeinheit.

Strom-Klassen (1)

- Ein Datenstrom wird durch eine Klasse repräsentiert.
  - Systemabhängigkeit wird in wenigen Klassen isoliert.
  - Rest des Programms bleibt systemneutral.
- Es gibt zwei Arten von Streams:
  - Byteströme – enthalten eine Folge von Bytes (z.B. pdf, Bilder, etc.)
  - Zeichenströme - enthalten Texte in Unicode
- Es gibt jeweils eine abstrakte Klasse für die vier Möglichkeiten aus Ein- und Ausgabe sowie Byte- und Zeichenströme.
  - Alle Byteströme werden abgeleitet von `InputStream` oder `OutputStream`.
  - Alle Zeichenströme werden abgeleitet von `Reader` oder `Writer`.
  - Diese Klassen geben die API vor und liefern eine partielle Implementierung .

	Byte	**Zeichen**
Eingabe	`InputStream`	`Reader`
Ausgabe	`OutputStream`	`Writer`

Strom-Klassen (2)

- `Reader` und `InputStream` definieren ähnliche APIs für unterschiedliche Datentypen.
- Ähnliches gilt für `Writer` und `OutputStream`.
- Für Byteströme und Zeichenströme gibt es jeweils zwei Arten:
  - Eingabe- oder Ausgabeströme
    - Sind mit bestimmten Datenquellen oder Datensenken verbunden.
    - Lesen von Datenquellen bzw. Schreiben in Datensenken!
  - Filterströme
    - Sind nicht direkt mit einer Quelle oder Senke verknüpft.
    - Sind mit einem Datenstrom verbunden.
    - Agieren wie ein Filter vor diesem Datenstrom.
    - Vorverarbeiten/Nachbearbeiten von Daten!
- Ströme können konkateniert werden.
  - Der Output eines Stroms ist der Input eines anderen Stroms.

Klasse InputStream

- Zum Lesen von Bytes.
- Auszug einiger Methoden:

  `int read()`
  - Liest das nächste Byte des Eingabestroms und gibt es als Wert vom Typ `int` zurück.
  - Der Rückgabewert -1 zeigt das Ende des Eingabestroms an.

  `byte[] readAllBytes()`
  - Liest alle Bytes aus dem Eingabestrom.

  `void close()`
  - Schließt den Eingabestrom.

Klasse Reader

- Zum Lesen von 16 Bit Zeichen.
- Auszug einiger Methoden

`int read()`

  - Liest das nächste Zeichen des Eingabestroms und gibt es als Wert vom Typ `int` zurück.
  - Der Rückgabewert -1 zeigt das Ende des Eingabestroms an.

`int read(char[] cbuf)`

  - Liest maximal `cbuf.length` Zeichen ein, speichert sie im Array `cbuf` und gibt die Anzahl an gelesenen Zeichen als Rückgabewert zurück.

`void close()`

  - Schließt den Eingabestrom.

Klasse OutputStream

- Zum Schreiben von Bytes.
- Auszug einiger Methoden:

`int write(int b)`
  - Schreibt das übergebene Byte `b` in den Ausgabestrom.

`void write(byte[] b)`
  - Schreibt das übergebene Bytearray `b` in den Ausgabestrom.

`void flush()`
  - Erzwingt das Schreiben von möglicherweise zwischengespeicherten Daten.

`void close()`
  - Schließt den Ausgabestrom.

Klasse Writer

- Zum Schreiben von 16 Bit Zeichen.
- Auszug einiger Methoden

`int write(int c)`

- Schreibt das übergebene Zeichen `c` in den Ausgabestrom.

`void write(char[] cbuf)`

- Schreibt das übergebene `char`-Array `cbuf` in den Ausgabestrom.

`void flush()`

- Erzwingt das Schreiben von möglicherweise zwischengespeicherten Daten.

`void close()`

- Schließt den Ausgabestrom.

Übersicht Strom-Klassen (1)

Anwendung	Bytestrom	Zeichenstrom

Hauptspeicher	`ByteArrayInputStream`
	`ByteArrayOutputStream`
	-
	-
		`CharArrayReader`
		`CharArrayWriter`
		`StringReader`
		`StringWriter`

Pipe	`PipedInputStream` `PipedOutputStream`	`PipedReader` `PipedWriter`
File	`FileInputStream` `FileOutputStream`	`FileReader` `FileWriter`
Filterströme (Filtering)	`FilterInputStream` `FilterOutputStream`	`FilterReader` `FilterWriter`
Zählstreams (für Zeilennummern)	`LineNumberInputStream`	`LineNumberReader`
Gelesene Daten zurücklegen	`PushbackInputStream`	`PushbackReader`
Ausgabe (Printing)	`PrintStream`	`PrintWriter`
Pufferung	`BufferedInputStream` `BufferedOutputStream`	`BufferedReader` `BufferedWriter`

Übersicht Strom-Klassen (2)

Anwendung	Bytestrom	Zeichenstrom
Konkatenation mehrerer InputStreams	`SequenceInputStream`	-
Objektserialisierung	`ObjectInputStream` `ObjectOutputStream`	- -
Datenkonvertierung (Standardtypen in rechnerunabhängiges Binärformat)	`DataInputStream` `DataOutputStream`	- -
Konvertierung zwischen Byte- und Datenströmen	- -	`InputStreamReader` `OutputStreamWriter`

Datenpufferung (1)

- Inputpuffer
  - Speichert schon gelesene aber nicht verarbeitete Daten.
    - Nicht Zeichen für Zeichen lesen und an das Programm übergeben.
    - Folgen von Zeichen in Puffer kopieren und bei einer Leseoperation an das Programm übergeben.
- Outputpuffer
  - Speichert temporäre Daten, die geschrieben werden.
  - Schreiben, wenn der Puffer voll wird (oder `flush()` aufgerufen wird).
- Viele I/O - Geräte unterstützen Operationen auf Blöcke.
- Der Datentransfer zwischen Puffer und Objekten ist normalerweise schnell.
- Die Pufferung verbessert die Laufzeit von Programmen.
  - Realisierung über Konkatenation

Datenpufferung (2)

- Beispiel: `BufferedReader` konkateniert mit `FileReader`
- Beispieldeklaration

```
BufferedReader inputStream = Files.newBufferedReader(Path.of("in.txt"));
```

  - Verbindet intern einen `BufferedReader` mit einem `FileReader`.
- Datenfluss
  - Textdatei → `FileReader` → `BufferedReader` → Speicher

src/at/ac/uibk/pm/streams/BookReaderApplication.java

Exception Handling: try with Resources (1)

- Die Verwaltung (vor allem das Schließen) von Ressourcen ist eine fehlerträchtige und komplizierte Angelegenheit.
  - z.B. kann beim Schließen in der `finally`-Klausel der `close`-Aufruf zu einer neuen Ausnahme führen.
- Ab Java 7 wurde genau diese Verwaltung vereinfacht.
- Bei einem try-Block werden die verwendeten Ressourcen direkt am Anfang angegeben (mit dem try-Block verbunden).
  - Am Ende der try-Anweisung werden die Ressourcen automatisch geschlossen.
  - Mehrere Ressourcen werden mit einem Strichpunkt getrennt angegeben.
  - Die Ressourcen müssen das Interface `AutoCloseable` implementieren.
    - Enthält eine Methode `close()`.
    - Alle Streams implementieren dieses Interface.
    - Eigene Ressourcen-Klassen können (sollten) dieses Interface auch implementieren.
- Seit Java 9 können Ressourcen auch außerhalb des try-Blocks definiert werden, wenn sie effektiv final sind.

Exception Handling: try with Resources (2)

```
Path inputFile = Path.of("src", "resources", "alice_in_wonderland.txt");
BufferedReader stream = null;
try {
    stream = Files.newBufferedReader(inputFile);
    System.out.println(stream.readLine());
} catch (IOException e) {
    // handle error, rethrow error or use throws instead of this catch!
} finally {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException ignored) {
        }
    }
}
```

```
Path inputFile = Path.of("src", "resources", "alice_in_wonderland.txt");
try (BufferedReader stream = Files.newBufferedReader(inputFile)) {
    System.out.println(stream.readLine());
} catch (IOException e) {
    // handle error, rethrow error or use throws instead of this catch!
}
```

Scan & Format

Scanning und Formatting

- Lesen von Eingabedaten
  - Programm zerlegt dann Eingabedaten für die Weiterverarbeitung.
- Formatierte Ausgabe
  - Programm erzeugt formatierte Ausgabe.
  - Beispiele:
    - `StringBuilder`
    - Bildschirm/Konsolenausgabe (`System.out` / `System.err` / `System.console()`)
    - Dateien und Pipes
    - …
- Java I/O bietet dazu zwei APIs an.
  - Scanner-API zum Einlesen und zerlegen in Token.
  - Formatter-API für die formatierte Ausgabe.

Standard I/O

- Einfachster I/O - Mechanismus
  - Texteingabe von der Tastatur
  - Textausgabe auf den Bildschirm ("Konsole")
- Auf praktisch jedem System verfügbar, minimale Systemanforderungen.
- Standard - I/O über vordefinierte Objekte
  - `System.in` Standard-Eingabe (← Tastatur)
  - `System.out` Standard-Ausgabe (→ Konsole)
  - `System.err` Fehlerausgabe (→ Konsole)
- `in`, `out`, `err`
  - öffentliche Klassenvariablen der Klasse `System`.
  - `in` ist ein `InputStream`.
  - `out` und `err` sind `PrintStreams`.

Scanner

- Scanner wird für das Zerlegen des Inputs verwendet.
  - Input wird in Tokens zerlegt.
  - Tokens werden dann entsprechend weiterverarbeitet.
  - Seit Java 1.5 (früher StringTokenizer-Klasse für Strings)
- Auszug einiger Methoden:

  `hasNext`, `hasNextLine` , `hasNextInt`, `hasNextDouble`…
  - Überprüfen ob noch ein Token (String), ein int-Wert, ein double-Wert etc. in der Quelle vorhanden ist.

  `next`, `nextLine` , `nextInt`, `nextDouble` ….
  - Auslesen des nächsten Tokens (String), int-Werts, double-Werts etc.

  `skip`
  - Überspringen des gegebenen String bzw. Regular Expression-Patterns.

  `findAll`
  - Finden aller Übereinstimmungen des String bzw. Regular Expression-Patterns und Rückgabe als geöffnete Streampipeline.

  `findInLine`, `findWithinHorizon`
  - Finden der nächsten Übereinstimmung des String bzw. Regular Expression-Patterns und Rückgabe als String.

Scanner – System.in

```
Scanner inputReader = new Scanner(System.in);
System.out.println("Please enter your name:");
String name = inputReader.nextLine();
System.out.println("Please enter your age: ");
int age = inputReader.nextInt();
System.out.println("Hello, %d year old %s.", age, name);
```

Scanner – Path

```
try (Scanner numbers = new Scanner(Path.of("us_numbers.txt"))) {
    numbers.useLocale(Locale.US);
    double sum = 0;
    while (numbers.hasNext()) {
        if (numbers.hasNextDouble()) {
            sum += numbers.nextDouble();
        } else {
            numbers.next();
        }
    }
    System.out.println(sum);
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Eingabedatei us_numbers.txt:

```
1000
1 1
1,500
1.1
```

Ausgabe:

```
2503.1
```

Formatierung

- Klassen, die Formatierung berücksichtigen
  - `PrintWriter`
  - `PrintStream`
- `System.out` und `System.err` sind `PrintStream`-Exemplare.
- Zwei Stufen der Formatierung werden unterstützt.
  - `print` und `println` formatieren individuelle Werte in der Standardform.
  - `format` formatiert jeden Wert anhand eines Formatierungsstrings.
    - Ist äquivalent zur `printf`-Methode.
    - Etwas strikter als in C!

Formatspezifizierer (Wiederholung)

- Formatspezifizierer sind wie folgt aufgebaut:

  `%[argument_index][flags][width][.precision]converter`

  `[argument_index]`
  - Position in der Argumentenliste (erstes Arg mit `1$`, zweites Arg mit `2$`).
  - Das vorherige Argument kann mit `<` referenziert werden.

  `[flags]`
  - Menge von Zeichen zur Modifikation der Ausgabe (beispielsweise `0`: führende 0en, `+`: Vorzeichen).

  `[width]`
  - Minimale Anzahl der Zeichen, die ausgegeben werden.

  `[.precision]`
  - Maximale Anzahl der Zeichen oder Genauigkeit bei Gleitkommazahlen.

  `converter`
  - Zeichen, das die Formatierung des Arguments bestimmt - erlaubte Formatierungen hängen vom jeweiligen Datentyp ab.
  - Ist als einziger Teil nicht optional.

- Mehr Details in der Dokumentation

Serialisierung

Java Serialisierung (1)

- Der Zustand eines Objekts wird so abgespeichert, dass er wieder rekonstruiert (ausgelesen) werden kann.
  - Serialisierung
    - Objekt wird als Folge von Bytes in einen Strom geschrieben.
  - Deserialisierung
    - Objekt wird als Folge von Bytes aus einem Strom gelesen.
- Objekt existiert über die Ausführung eines Programms hinaus → Speicherung und Austausch.
- Objekt wird mit all seinen Komponenten in den Strom geschrieben, d.h. Referenzen werden **auch** serialisiert.

src/at/ac/uibk/pm/serialization/ListSerializerApplication.java

Java Serialisierung (2)

- Deserialisieren von nicht vertrauenswürdigen Datenquellen sollte unbedingt vermieden werden.
- Nachteile der Serialisierung/Deserialisierung:
  - Verringerung der Flexibilität.
    - Datenkapselungs- und das Informationhiding-Prinzip wird aufgeweicht.
    - Für die default-serialisierte Form einer Klasse werden private und package-private Objektvariablen Teil der öffentlichen API der serialisierten Klasse.
  - Die `readObject` Methode ist eine Art Konstruktor die Exemplare aller serialisierbaren Objekte erzeugen kann.
  - Bietet eine große Fläche für Angreifer. Beispiele:
    - Remote Code Execution (RCE)
    - Denial-of-Service (DoS) Angriffe durch Deserialisierungsbomben.

Alternativen zu Java Serialisierung

- Es existieren verschiedene Alternativen um Objekte in Byte-Sequenzen umzuwandeln und Byte-Sequenzen zurück in Objekte zu transformieren.
- Führende Cross-Plattform Repräsentationen strukturierter Daten sind JSON und Protobuf.
- Ist das Ausweichen auf Alternativen nicht möglich:
  - Niemals Daten von nicht vertrauenswürdigen Quellen deserialisieren.
  - Deserialisierungsfilter durch java.io.ObjectInputFilter umsetzen:
    - Blacklisting (Akzeptieren aller Klassen außer den Klassen auf der Blacklist)
    - Whitelisting (nur die Klassen auf der Whitelist werden akzeptiert)
  - Gut durchdachte serialisierte Form der Klasse verwenden.

Java Serialisierung - Voraussetzung

- Ein Objekt kann nur serialisiert werden, wenn die entsprechende Klasse das Interface `Serializable` implementiert.
  - Sonst: `java.io.NotSerializableException`
- `Serializable` ist ein Marker-Interface (leeres Interface)!
- Bei der Serialisierung werden geschrieben
  - Klasse
  - Klassensignatur
  - Werte der **nicht statischen** und **nicht transienten** Elemente (auch Referenzen).

Anpassen der Serialisierung bei Klassen

- Serialisierung/Deserialisierung kann angepasst werden.
- Implementieren der Methoden

  `writeObject`
  - Für das Schreiben eines Objekts (Anpassen, zusätzliche Informationen etc.).

  `readObject`
  - Für das Lesen eines Objekts (Lesen, Anpassen, Updates etc.).
- Form der `writeObject`-Methode

```
private void writeObject(ObjectOutputStream s) throws IOException {
    // Benutzerspezifische Anpassung …
}
```

- Form der `readObject`-Methode

```
private void readObject(ObjectInputStream s) throws IOException,
        ClassNotFoundException {
    // Benutzerspezifische Anpassung …
    // Zusätzliche Updates …
}
```

transient

- Oft soll ein Teil eines Objekts nicht serialisiert werden.
- Schlüsselwort `transient` bei der Deklaration von Objektvariablen verwenden.
  - Die entsprechende Objektvariable wird **nicht** serialisiert.
  - Beim Einlesen wird die entsprechende Objektvariable mit dem Defaultwert initialisiert.
  - Wenn das nicht gewünscht wird – überschreiben von `readObject` mit einer entsprechenden Initialisierung.

Serialisierung unterbinden

- Die Serialisierung kann in einer Subklasse durch Implementieren der Methoden `writeObject` und `readObject` und Werfen einer Exception unterbunden werden.
- Beispiel:

```
public class A implements Serializable {
    …
}
```

```
public class B extends A {
    …
    private void writeObject(ObjectOutputStream s) throws IOException {
        throw new NotSerializableException();
    }

    private void readObject(ObjectInputStream s) throws IOException,
            ClassNotFoundException {
        throw new NotSerializableException();
    }
}
```

Serialisierung und Vererbung

- Ist eine Basisklasse nicht serialisierbar, kann die Subklasse durch selbst Implementierte `writeObject` und `readObject` Methoden alle zugänglichen Variablen beim Serialisieren auslesen und beim Deserialisieren wiederherstellen.
- Dazu muss in der abgeleiteten Klasse der Zugriff auf einen Parameterlosen Konstruktor der Basisklasse möglich sein!
  - Dieser wird zum Erstellen eines Objekts der Basisklasse verwendet.
  - Wenn dieser nicht vorhanden ist kommt es zu einem Laufzeitfehler.

Versionsverwaltung (1)

- Verschiedene Versionen einer Klasse können Probleme bereiten.
  - Objekt wird serialisiert.
  - Klasse wird verändert.
    - Objektvariablen löschen
    - Typen ändern
  - Deserialisieren funktioniert dann nicht mehr (`InvalidClassException`).
- Erkennung in Java
  - Bei der Serialisierung wird eine eindeutige Kennung geschrieben (UID).
    - Hashcode aus Namen, Objektvariablen, Parametern, Sichtbarkeit usw.
  - Ändert sich eine Klasse, dann ändert sich der Hashcode.
    - Keine Kompatibilität mehr!

Versionsverwaltung (2)

- SUID (serialVersionUID)
- Beispiel:

```
private static final long serialVersionUID = 902774526861406703 5L;
```

- Benutzer kann die SUID selbst berechnen.
  - Händisch
    - Aufruf des Hilfsprogramms `serialver` mit Klassenname.
    - Ausgabe des Programms in das eigene Programm kopieren.
  - IDE unterstützt die Generierung.
- Es kann eine eigene SUID angegeben werden (z.B. 1L).
- Bestimmte Änderungen (nicht alle) können vorgenommen werden, wenn die SUID gleich bleibt.
  - Neue Objektvariablen werden beim Einlesen standardmäßig mit den default-Werten für ihren Typ initialisiert.
  - Fehlende Objektvariablen werden ignoriert.
- Bei inkompatiblen Änderungen muss die SUID verändert werden.
  - Neu generieren oder eigene hochzählen.

Quellen

- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Hanspeter Mössenböck: **Sprechen Sie Java?**, dpunkt.verlag, 5. Auflage, 2014
- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018