Dateien und Verzeichnisse

Motivation

- Verzeichnisse durchsuchen und erstellen
- Dateien suchen
- Eigenschaften von Dateien ermitteln
- Dateien kopieren, verschieben und löschen
- Textdateien lesen und schreiben

$\Rightarrow$ `java.io, java.nio`

Die wichtigsten Klassen(C) und Schnittstellen(I) zur Bearbeitung von Dateien und Verzeichnissen

- `BasicFileAttributes(C)`: Beschreibt die Eigenschaften einer Datei
- `BufferedReader(C)`: Liest Textdateien effizient
- `BufferedWriter(C)`: Schreibt Textdateien effizient
- `DirectorySteam(I)`: Aufzählung (Collection) von Verzeichnissen und Dateien
- `Files(C)`: Sammlung statischer Methoden, um Verzeichnisse und Dateien zu bearbeiten
- `IOException(C)`: Beschreibt einen IO-Fehler
- `Path(I)`: Repräsentiert einen Datei- oder Verzeichnisnamen
- `Paths(C)`: Statische Methoden, um `Path`-Objekte zu erzeugen

Fehlerabsicherung

- Was ist, wenn der Datenträger voll ist?
- Wenn der Benutzer einen USB-Stick im laufenden Betrieb einfach entfernt?
- Wenn eine Datei zwar vorhanden ist, das Java-Programm aber unzureichende Rechte hat, sie zu lesen?

$\Rightarrow$ IOException fangen

$\Rightarrow$ Spezialisierungen sind z.B. EOFException, FileNotFoundException

Ressourcen schließen

- Für viele IO-Klassen gibt es die Methode close
- Objekte sollten „geschlossen“ werden, um nicht unnötig Ressourcen zu blockieren

$\Rightarrow$ Ressourcenverwaltung mit Fehlerabsicherung kombinieren

```
try (IOClass1 ioob1 = ...; IOClass2 ioob2 = ...){
  // IO-Objekte behandeln
} catch (IOException) {
  // Fehlerabsicherung
}
```

Wiederholung: Was sind Ressourcen?

Alle Klassen bzw. Objekte die das Interface java.lang.AutoCloseable implementieren

Ohne ARM (vor Java 1.7)

```
public String methodXY() {
  BufferedReader reader = new
      BufferedReader(...);
  try {
    return reader.readLine();
  }
  finally {
    if (reader != null)
      reader.close();
  }
}
```

Mit ARM (ab Java 1.7)

```
public String methodXY() {
  try (BufferedReader reader = new
      BufferedReader(...)) {
    return reader.readline();
  }
}
```

Dateien und Verzeichnisse ergründen

Informationen über Dateien und Verzeichnisse

- Existiert ein Verzeichnis?
- Welche Daten sind drin?
- Wie groß sind die Dateien?
- Was darf verändert werden?

Aktuelles Verzeichnis, Home und des aktuellen Benutzers ermitteln

```
String current = System.getProperty("user.dir");
String home = System.getProperty("user.home");
String tmp = System.getProperty("java.io.tmpdir");
```

Die „Path“-Schnittstelle

- Die Schnittstelle `Path` aus dem Paket `java.nio.files` repräsentiert einen Verzeichnis- bzw. Dateinamen
- Mit `get` absoluten oder relativen Pfad bekommen
- Achtung: beim Erzeugen eines Path-Objekts wird nicht überprüft, ob es die Datei oder das Verzeichnis wirklich gibt

Aktuelles Verzeichnis, Home und des aktuellen Benutzers ermitteln

```
import java.nio.file.*;
...
String home = System.get Property("user.home");
Path p = Paths.get(home);
```

Beispiele für `get`

```
public static void main(String[] args) {
  String home = System.getProperty("user.home");
  Path txt = Paths.get(home, "meintext.txt");
  Path foto = Paths.get(home, "Pictures", "foto.jpg");
  System.out.println(foto.toString());
  System.out.println(txt.toString());
}
```

`resolve`-Methode

Um ausgehend von einem vorhandenen `Path`-Objekt ein neues zu bilden, `resolve`-Methode verwenden

```
String home = System.getProperty("user.home");
Path p = Paths.get(home)
Path datei = p.resolve("datei.tmp");
```

`relativize`-Methode

`path1.relativize(path2)` bildet für `path2` den relativen Pfad, der von `path1` ausgeht

```
Path verz = Paths.get("a", "b", "c");
Path datei = Paths.get("a", "b", "c", "d", "e", "f.tmp");
Path r = verz.relativize(datei);
System.out.println(r);

// Ausgabe: d/e/f.tmp
```

Wichtig

- Vermeiden Sie absolute, betriebssystemspezifische Pfadangaben!
- `filename = "C:\\verz\\name.txt"`

$\Rightarrow$ Programm nicht mehr plattformunabhängig

Methoden der `Path`-Schnittstelle

- `isAbsolute` testet, ob es sich um einen absoluten oder einen relativen Pfad handelt
- `toAbsolute` erzeugt ggf. aus einem relativen ein neues absolutes `Pfad`-Objekt
- `toRealPath` ruft ebenfalls zuerst `toAbsolute` auf, ersetzt dann aber Kurzschreibweisen wie .. für das übergeordnete Verzeichnis und . für das aktuelle Verzeichnis durch die tatsächlichen Verzeichnisnamen, passt die Groß- und Kleinschreibung an (Windows) und berücksichtigt Links
- `toString` gibt den Dateinamen in der Notation des Betriebssystems zurück
- `getFileName` liefert ein neues `Pfad`-Objekt des eigentlichen Dateinamens ohne Pfad
- `getParent` liefert ein `Pfad`-Objekt des Verzeichnisses, in dem sich der letzte Teil des ursprünglichen `Pfad`-Objekts befindet

Testen, ob ein Verzeichnis bzw. eine Datei vorhanden ist

Wird fehlerfrei ausgefuehrt, obwohl nicht vorhanden

```
Path p = Paths.get(home, "meinverz", "bla.txt");
```

Überprüfung

```
if (Files.exists(p){ ...}
```

Statische Methoden der `Files`-Klasse

- `exists` Existiert die Datei oder das Verzeichnis
- `isDirectory` Handelt es sich um ein Vezeichnis?
- `isRegularFile` Ist es eine gewöhnliche Datei?
- `isReadable` Kann die Datei gelesen werden?
- `isWriteable` lKann die Datei verändert werden
- `isHidden` Ist die Datei verborgen?
- `isExecutable` Ist die Datei ausführbar?

Eigenschaften einer Datei ermitteln

- `size` liefert die Grüße der Datei in Bytes
- `getLastModifiedTime` ermittelt den Zeitpunkt der letzten Änderung
- `getOwner` verrät, wem die Datei gehört.

⇒ Die genannten Methoden können IOExceptions auslösen und müssen daher mit try-catch abgesichert werden

Beispiel

```
if (Files.exists(p)) {
  try {
    System.out.println("Dateiname: " + p.toAbsolutePath());
    System.out.println("Groesse    : " + Files.size(p));
    System.out.println("Besitzer:  " + Files.getOwner(p).getName());
    var changeInst = Files.getLastModifiedTime(p).toInstant();
    var changeLdt = LocalDateTime.ofInstant(changeInst, ZoneId.systemDefault());
    System.out.println("Letzte Aenderung: " + changeLdt);
  } catch (IOException ioex) {
    System.out.println("Fehler: " + ioex.getMessage());
  }
}
```

Liste der Dateien in einem Verzeichnis ermitteln

- Z.B. nur *.pdf anzeigen lassen
- statische Methode `newDirectoryStream` der `Files`-Klasse. Sie erzeugt ein `DirectoryStream<Path>`-Objekt

Variante 1

```
Path phome = Paths.get(home);
try (DirectoryStream<Path> files = Files.newDirectoryStream(phome)) {
  for (Path found : files)
  if (Files.isRegularFile(found) && found.toString().toLowerCase().endsWith(".pdf"))
  System.out.println(found);
} catch (IOException e) {
  e.printStackTrace();
}
```

Liste der Dateien in einem Verzeichnis ermitteln

Variante 2

```
Path phome = Paths.get(home);
try (Stream<Path> st = Files.list(phome)) {
  st.filter(pth -> Files.isRegularFile(pth)).filter(pth -> pth.toString().toLowerCase().
      endsWith(".pdf")).forEach(System.out::println);
} catch (IOException e) {
  e.printStackTrace();
}
```

Verzeichnisse und Dateien bearbeiten

- `createDirectory`: Verzeichni anlegen
- `createDirectories`: Verzeichnis samt Unterverzeichnis erzeugen
- `createFile`: neue, leere Datei erzeugen
- `createTempDirectory`: temporäres Verzeichnis erzeugen
- `createTempFile`: temporäre Datei erzeugen
- `copy`: kopieren
- `move`: Datei oder Verzeichnis verschieben
- `delete`: Datei oder Verzeichnis löschen
- `deleIfExists`: Datei oder Verzeichnis löschen

Beispiel

```
public static void main(String[] args) {
  String home = System.getProperty("user
      .home");
  Path v1 = Paths.get(home, "verz1");
  Path v23 = Paths.get(home, "verz2", "
      unterverz3");
  Path v23neu = Paths.get(home, "verz2",
       "uv3-neu");
  Path d1 = v1.resolve("datei1.tmp");
  Path d2 = v1.resolve("datei2.tmp");
```

```
Verzeichnisse und Dateien in Ihrem
    Heimatverzeichnis:
verz1
  datei1.tmp
  datei2.tmp
verz2
  uv3-neu
```

```
  try {
    if (!Files.exists(v1))
      Files.createDirectory(v1);
    if (!Files.exists(v23))
      Files.createDirectories(v23);
    if (Files.exists(d1))
      Files.delete(d1);
    Files.createFile(d1);
    Files.copy(d1, d2,
        StandardCopyOption.
        REPLACE_EXISTING);
    if (!Files.exists(v23neu))
      Files.move(v23, v23neu);
  } catch (IOException e) {
    e.printStackTrace();
}
```

Textdateien lesen und schreiben

- `lines`: Liefert einen `String`-Stream mit dem zeilenweisen Inhalt der Datei
- `newBufferedReader`: Liefert ein `BufferedReader`-Objekt zum Auslesen der Datei
- `newBufferedWriter`: Erzeugt ein `BufferedWriter`-Objekt zum Schreiben in eine Datei
- `readAllLines`: Liefert eine Liste von `Strings` mit den Zeilen der Datei
- `write`: Schreibt Textzeilen in eine Datei

Textdateien schreiben

- Für das effiziente Schreiben in Textdateien sieht Java die Klasse `BufferedWriter` vor
- `BufferedWriter` verwaltet einen Pufferspeicher, um Ausgaben zwischenzuspeichern und so die Datenträgerzugriffe zu minimieren.
- Mit `write` kann auf dem `BufferedWriter` geschrieben werden

Beispiel

Aufgabe

- Datei `test.txt`, wenn diese Datei noch nicht existiert, anlegen
- Wenn vorhanden, Text ergänzen
- In die Datei: Aktuelles Datum und Uhrzeit, „Hello BufferedWriter“, „Ende“ schreiben

```
String current = System.getProperty("
    user.dir");
Path txtfile = Paths.get(currrent, "text
    .txt");
try (var bw = Files.newBufferedWriter(
    txtfile, StandardOpenOption.APPEND,
    StandardOpenOption.CREATE)){
  bw.write(LocalDateTime.now()+"\n");
  bw.write("Hello BufferedWriter\n");
  bw.write("Ende\n");
} catch (IOException ex) {}
 ex.printStackTrace();
}
```

Alternative zu BufferedWriter

write-Methode der Files-Klasse

```
// Forsetzung Beispiel
String[] txt = {"Noch", "mehr", "Text"};
var txtlist = Arrays.asList(txt);
try{
  Files.write(txtfile, txtlist, StandardOpenOption.APPEND);
} catch (IOException e) {
  e.printStackTrace();
}
```

Textdateien auslesen

```
// Forsetzung Beispiel
try (var br = Files.newBufferedReader (txtfile)) {
  String line;
  while ((line = br.readlLine ())  != null )
    System.out.println(line);
} catch (IOException e) {
  e.printStackTrace();
}
```

Ganze Datei auslesen

```
// Forsetzung Beispiel
try {
  List<String> all = Files.readAllLines(txtfile);
  all.stream().forEach(System.out::println);
} catch (IOException e) {
  e.printStackTrace();
}
```

. . . oder mit Lambda-Ausdrücken

```
// Forsetzung Beispiel
try {
  Files.lines(textfile).limit(3).forEach(System.out::println);
} catch (IOException e) {
  e.printStackTrace();
}
```

Serialisierung

Motivation: Objektserialisierung

- Objekte existieren nur im Speicher, solange Programm läuft
- Eine längere Speicherung ist nur in beständigen sog. „persistenten“ Datenspeichern möglich (z.B. Dateien im Dateisystem)
- Eine Speicherung von Objekten im Textformat über `toString`-Methode ist jedoch NICHT geeignet
  - Struktur (z.B. Vererbungshierarchie, Attributtypen) geht verloren
  - Inhalte gehen verloren, wenn nicht von `toString` ausgegeben
  - Die in Attributen gehaltenen Referenzen auf andere Objekte gehen unter Umständen verloren
  - Ein Entwickler muss sich ein Format zur Speicherung ausdenken UND dieses Format mit ALLEN anderen Entwicklern vereinbaren

Motivation: Objektserialisierung

besser: Speicherung in wohldefiniertem Byte-Format

- $\Rightarrow$ Struktur, Inhalte und verknüpfte Objekte vollständig rekonstruierbar
- $\Rightarrow$ Speichern durch Objekt-Serialisierung
- $\Rightarrow$ Laden durch Objekt-Deserialisierung

Basis: Objektstreams

Serialisierung

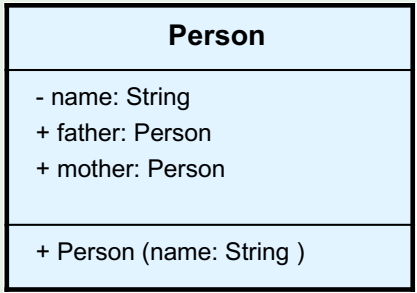
- Konstruktor: `ObjectOutputStream(OutputStream out)`
- Schreiben primitiver Typen: `void writeChar(int v), void writeInt(int v), void writeFloat(float v) ...`
- Schreiben von Referenztypen: `void writeObject(Object v)`

Deserialisierung

- Konstruktor: `ObjectInputStream(InputStream in)`
- Lesen primitiver Typen: `char readChar(int v), int readInt(int v), float readFloat(float v) ...`
- Schreiben von Referenztypen: `Object readObject(Object v)`

Beispiel: Stammbaum

```
public class StammbaumSerialisierung {
  public static void main(String[] args)
      {
    Person opa = new Person("Eugen");
    Person oma = new Person("Therese");
    Person vater = new Person("Barny");
    Person mutter = new Person("Wilma");
    Person kind1 = new Person("Fritzchen
        ");
    Person kind2 = new Person("Kalli");
    vater.father = opa;
    vater.mother = oma;
    kind1.father = kind2.father = vater;
    kind1.mother = kind2.mother = mutter
        ;
```



Beispiel: Serialisierung von Objekten

```
// ObjectOutputStream kann Objekte in Bytes wandeln
// FileOutputStream kann Bytes in eine Datei schreiben

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("family.ser")))
    {
  oos.writeObject(kind1);
  oos.writeObject(kind2);
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

Beispiel: Deserialisierung von Objekten

```
// FileInputStream kann Bytes aus einer Datei lesen
// ObjectInputStream kann Bytes in Objekte wandeln

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("family.ser"))) {
  kind1 = (Person) ois.readObject();
  kind2 = (Person) ois.readObject();
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
} catch (ClassNotFoundException e) {
  e.printStackTrace();
}
```

Bedingungen zur Objektserialisierung

- Die Klasse muss Interface `Serializable` implementieren
- die meisten Klassen der Klassenbibliothek sind serialisierbar
- nicht serialisierbar sind i.d.R. Klassen, die
  - flüchtige Daten enthalten oder
  - Strukturen modellieren, die sich nicht rekonstruieren lassen
- Einzelne flüchtige Attribute von prinzipiell serialisierbaren Klassen sind mit dem Schlüsselwort `transient` von der Serialisierung ausnehmbar

Bedingungen zur Objektserialisierung

- Bei der Übergabe an `writeObject` werden automatisch gespeichert:
  - Werte der nicht-statischen und nicht-transienten Attribute
  - referenzierte Objekte (rekursiv)
- Alternativ ist (De-)Serialisierung selbst implementierbar:
  - `private void writeObject(ObjectOutputStream oos)`
  - `private void readObject(ObjectInputStream ois)`
- Lesen muss in gleicher Reihenfolge wie Schreiben erfolgen

Bedingungen zur Objektserialisierung

- Bei der (De-)Serialisierung wird nicht der Konstruktor des erzeugten Objekts aufgerufen.
- Ausnahme:
  - Serialisierbare Klasse, mit Superklasse(n), die nicht das Interface `Serializable` implementieren.
  - Der parameterlose Konstruktor der nächsthöheren nicht-serialisierbaren Vaterklasse wird aufgerufen.
  - Die aus der nicht-serialisierbaren Vaterklasse geerbten Membervariablen werden nicht automatisch gespeichert.
  - Auf diese Weise soll sichergestellt werden, dass sie sinnvoll initialisiert werden.
- Initialisierungscode im Konstruktor wird nicht ausgeführt. Variablen ggf. mit Standardwerten initial belegt

Objektserialisierung: Probleme

- Die Serialisierung von Objekten ist nicht trivial, da Objekte Attribute enthalten können, die selbst wieder auf Objekte verweisen (Aufbau von Assoziationen)
- Folgende Probleme sind zu behandeln:
  - Verfolgung von Objektreferenzen
  - Korrekter Umgang mit zyklischen Referenzen
  - Mehrfach referenzierte Objekte

$\Rightarrow$ Der `ObjectOutputStream` hält zu diesem Zweck eine Hash-Tabelle, in der alle bereits serialisierten Objekte verzeichnet werden.

Objektserialisierung: Probleme

Bei der (De-)Serialisierung von Objekten können Konsistenzprobleme auftreten

Objektserialisierung: Probleme

Lösung: Versionierungsmechanismus

- Beim Serialisieren eines Objektes wird eine Versionsnummer für die zugehörige Klasse erzeugt und mit in die Ausgabedatei gespeichert.
- In die Versionsnummer gehen Name und Signatur der Klasse, implementierte Interfaces sowie Operationen und Konstruktoren ein.
- Beim Deserialisieren wird die, in der Datei gespeicherte, Versionsnummer mit der aktuellen Versionsnummer der Klasse verglichen. Stimmen beide nicht überein, so gibt es eine Ausnahme vom Typ `InvalidClassException` und der Deserialisierungsvorgang bricht ab.

Versionierung

Versionsnummer für Klasse

- Das Interface `Serializable` enthält eine `long`-Konstante `serialVersionUID`. Diese speichert eine Versionsnummer der Klasse: `static final long serialVersionUID = 42L;`
- IWird die Konstante nicht explizit gesetzt, so wird Sie von writeObject automatisch berechnet und in die Ausgabedatei geschrieben.

Versionierung

Versionsnummer für Klasse

- Diese Art der Versionierung ist zwar recht sicher, aber auch sehr rigoros.
- Schon eine kleine Änderung an der Klasse macht die serialisierten Objekte unbrauchbar, weil sie sich nicht mehr deserialisieren lassen.
- Wird beispielsweise eine neue Methode `public void test()` hinzugefügt (die für das Deserialisieren eigentlich völlig bedeutungslos ist), ändert sich die `serialVersionUID`

$\Rightarrow$ Datei lässt sich nicht mehr deserialisieren

$\Rightarrow$ Alternative: Konstante selbst (nach eigenen definierten Regeln) verwalten!