Dateiverarbeitung

1. Die Klasse File

Die Klasse java.io.File ist nicht nur für die Verwendung von Dateien zuständig, sondern auch für den Umgang mit Verzeichnissen.
Die Klasse »File« hat keinerlei Methoden zum Lesen und Schreiben von Dateien. Es kann nur geprüft werden, ob eine Datei existiert und welche Attribute sie besitzt (z.B. readonly).

Beispiel:

```
import java.io.File;
import java.io.IOException;

public class ErstelleDatei{
    public static void main(String[] args) {
        // Erzeugung unseres File-Objektes
        File beispielDatei = new File("C:/Beispiel.txt");
        /* Überprüfung, ob die mit dem Pfad assozierte Datei
           bzw. Verzeichnis existiert */
        if(!beispielDatei.exists()){
            try{
                // Erstelle Datei auf Festplatte
                boolean wurdeErstellt = beispielDatei.createNewFile();
                // Überprüfung, ob die Datei erstellt wurde
                if(wurdeErstellt)
                {
                    System.out.println("Beispiel.txt wurde erfolgreich" +
                            " auf dem Laufwerk c erstellt");
                }
                else
                {
                    System.out.println("Beispiel.txt wurde nicht" +
                                " auf dem Laufwerk c erstellt");
                }
            }
            catch (IOException ex) {
                // Ein Fehler ist aufgetreten.
                ex.printStackTrace();
            }
        }
    }
}
```

Abbildung 31: Beispiel File

Die Klasse File enthält u.a. statische Methoden zur Klärung grundlegender Fragen:

1. Existiert die Datei oder das Verzeichnis?
2. Ist es ein Verzeichnis? Ist die Datei readable?
3. Kann die Datei verändert werden?
4. Ist die Datei ausführbar?
5. Wann wurde die Datei zuletzt verändert?
6. Wem gehört die Datei?

Aufgabe:
Wenden Sie die Methoden der Klasse File an, um die oben genannten Fragen anhand selbstgewählter Beispiele zu beantworten! Drucken Sie die Ergebnisse auf der Konsole (vergleiche Beispiel)

```
Ordner daten wurde erfolgreich erstellt!
Beispiel.txt wurde erfolgreich erstellt!
Abs. Pfad:      Z:\000\JavaBFI12B\daten
Pfad:           Z:\000\JavaBFI12B\daten
Name:           daten
Existiert:      true
Ausführbar:     true
Lesbar:         true
Schreibbar:     true
Directory:      true
Datei:          false
LastMod:        2020-10-12T18:19:14.652
```

2. java.nio.file

Mit der Version 7 wurde in Java die Art und Weise, Dateien zu verwalten und Dateiinhalte zu bearbeiten, von Grund auf neu implementiert. Zwar existiert auch weiterhin die altbekannte Klasse java.io.File, zusätzlich gibt es nun jedoch im Package java.nio.file ein komplett neue Möglichkeit des Zugriffs auf das Dateisystem.
Um in Java-Versionen vor Java SE 7 mit Dateien zu arbeiten, muss die Klasse java.io.File genutzt werden. Exemplare dieser Klasse bilden jeweils den Java-Repräsentanten einer Datei im lokalen Dateisystem.
Anders ausgedrückt: Ein java.io.File-Objekt repräsentiert immer eine Datei im lokalen Dateisystem. Eine Möglichkeit, den Dateibegriff abstrakter und weiter zu interpretieren, ist nicht vorgesehen. Ein Dateizugriff auf ein entferntes System (beispielsweise per FTP) lässt sich mit java.io.File nicht realisieren und muss zwangsläufig mit externen Frameworks umgesetzt werden.

1. Anlegen eines Pfades:
   Um eine Datei schreiben zu können, muss zunächst ein Pfad erstellt werden. Dazu wird ein Path-Objekt benötigt.

```
// relative Referenz im aktuellen Verzeichnis
Path p1 = Paths.get("daten/gruppenbild.jpg");
// absolute Referenz im Dateisystem
Path p2 = Paths.get(URI.create("file:/C:/test.txt"));
// absolute Referenz zum Verzeichnis /klassen/bfi12a/AEuP
Path p3 = Paths.get("/klassen", "bfi12a", "AEuP");
```

Abbildung 32: Anlegen eines Pfades

2. Schreiben in eine Datei
   Zum Schreiben kann die Methode Files.write() verwendet werden. Mit dieser kann wird der Pfad und der zu schreibende Text als Bytes übergeben. Diese Art ist jedoch nicht unbedingt für größere Aufgaben geeignet.

```
Path path = Paths.get("src/main/resources/question.txt");
String question = "To be or not to be?";
Files.write(path, question.getBytes());
```

Abbildung 33: Schreiben in eine Datei

3. Das bessere Schreiben
   Bei größerem Schreibzugriff ist ein gepufferter Schreibzugriff besser geeignet. Dabei wird ein Charset festgelegt und ein String geschrieben. Hierfür muss im Gegensatz zu der ersten Variante nicht jedes Byte einzeln geschrieben werden.

```
Path path = Paths.get("src/main/resources/shakespeare.txt");
try(BufferedWriter writer = Files.newBufferedWriter(path, Charset.forName("UTF-8"))){
      writer.write("To be, or not to be. That is the question.");
}catch(IOException ex){
      ex.printStackTrace();
}
```

Abbildung 34: Schreiben mit Buffer

4. eine Datei kopieren
   Mit Java.nio kann auch der Inhalt einer Datei kopiert werden. Dazu werden zwei Pfade angelegt und beide der Methode Files.copy() übergeben.
5. Lesen einer Datei
   Um eine Datei zu lesen stehen zwei Möglichkeiten zur Auswahl. Entweder kann der gesamte Inhalt einer Datei auf einmal, oder zeilenweise gelesen werden.

```
Path path = Paths.get("D:/data/file.txt");
      try {
            byte[] bs = Files.readAllBytes(path);
            List<String> strings = Files.readAllLines(path);

            System.out.println("Read bytes: \n"+new String(bs));
            System.out.println("Read lines: \n"+strings);
      } catch (Exception e) {
            e.printStackTrace();
      }
```

Abbildung 35: Lesen einer Datei mit NIO

Ausgabe:

```
Read bytes:
Hello world
This is Read file example
Thank you
Read lines:
[Hello world, This is Read file example, Thank you]
```

Übung:

Erstelle zu jeder Möglichkeit eine Datei zu schreiben ein Programm:

a) 1.000.000 Zeichen mit Files.write() in eine lokale Datei schreiben
b) 1.000.000 Zeichen mit Buffered Writer in eine lokale Datei schreiben