

## File Services in Java

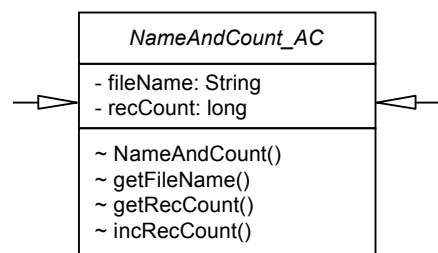
Es erscheint als sinnvoll, die Klassen der File Services „von hinten nach vorn“ zu zeigen, da es nicht möglich ist,

- die Klasse `FileRegister` ohne Kenntnis der array-beschreibenden Klassen
- die Front End-Klassen `InFileUtil` / `OutFileUtil` wiederum ohne `FileRegister`
- alle drei Klassen ohne `FS_Code`

zu verstehen. Daher beginnt die Auflistung mit `NameAndCount_AC`, gefolgt von `BR_RefNameAndCount`, `BW_RefNameAndCount`, `FS_Code` und `Trace`. Danach folgen die Klassen `FileRegister`, `InFileUtil` und `OutFileUtil`.

### 8.1 Klasse `NameAndCount_AC`

8



```

package FileService;

/**
 * The class NameAndCount_AC is the base class for the array entry classes
 * BR_RefNameAndCount and BW_RefNameAndCount.
 */
abstract class NameAndCount_AC
{
    private String fileName;
    private long    recCount;

    /**
     * constructor for objects of class NameAndCount_AC
     */
    NameAndCount_AC(String fileName)
    {
        this.fileName = fileName;
        this.recCount = 0;
    }

    String getFileName() { return this.fileName; }
  
```

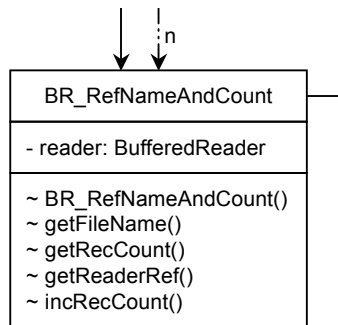
```

    long    getRecCount() { return this.recCount; }

    void    incRecCount() { this.recCount++; }
}

```

## 8.2 Klasse BR\_RefNameAndCount



```

package FileService;

import java.io.BufferedReader;

/**
 * The class BR_RefNameAndCount describes the data and the behaviour of an
 * input array entry within the FileRegister, using its base class.
 */
class BR_RefNameAndCount extends NameAndCount_AC
{
    private BufferedReader reader;

    /**
     * constructor for objects of class BR_RefNameAndCount
     */
    BR_RefNameAndCount(BufferedReader reader,
                       String fileName)
    {
        super(fileName);
        this.reader = reader;
    }

    BufferedReader getReaderRef() { return this.reader; }
}

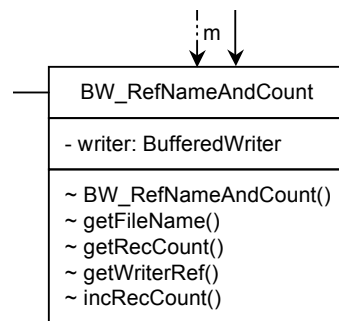
```

## Anmerkungen

Die Klasse `BR_RefNameAndCount` beschreibt zusammen mit ihrer Oberklasse `NameAndCount_AC` einen Array-Eintrag für eine Eingabedatei. Darin werden eine Referenz `reader` auf das zugehörige Zugriffsobjekt, der Dateiname und ein Zugriffszähler geführt. Der `super ( )`-Aufruf gibt den Dateinamen an die Oberklasse weiter, die ihn speichert und den Zugriffszähler initialisiert.

`BR_RefNameAndCount` erbt drei Methoden von `NameAndCount_AC`, denen es nur eine hinzufügt.

## 8.3 Klasse `BW_RefNameAndCount`



```

package FileService;

import java.io.BufferedWriter;

/**
 * The class BW_RefNameAndCount describes the data and the behaviour of an
 * output array entry within the FileRegister, using its base class.
 */
class BW_RefNameAndCount extends NameAndCount_AC
{
    private BufferedWriter writer;

    /**
     * constructor for objects of class BW_RefNameAndCount
     */
    BW_RefNameAndCount(BufferedWriter writer,
                       String      fileName)
    {
        super(fileName);
        this.writer = writer;
    }

    BufferedWriter getWriterRef() { return this.writer; }
}
  
```

## Anmerkungen

Die Klasse `BW_RefNameAndCount` beschreibt zusammen mit ihrer Oberklasse `NameAndCount_AC` einen Array-Eintrag für eine Ausgabedatei. Darin werden eine Referenz `writer` auf das zugehörige Zugriffsobjekt, der Dateiname und ein Zugriffszähler geführt. Der `super()`-Aufruf gibt den Dateinamen an die Oberklasse weiter, die ihn speichert und den Zugriffszähler initialisiert.

`BW_RefNameAndCount` erbt drei Methoden von `NameAndCount_AC`, denen es ebenfalls nur eine hinzufügt.

## 8.4 Klasse FS\_Code

```
package FileService;

/**
 * This enumeration defines the result codes of the calls to the file service
 * methods and associates them with the OK / error messages.
 * The positive code range is used for exceptions. The negative code range is
 * used for error messages.
 */
public enum FS_Code
{
    ERROR_LIMIT_EXCEEDED (255, "File I/O error limit exceeded"),
    INTERNAL_FATAL_ERROR ( 12, "Internal fatal error"),
    OUTPUT_CLOSE_FAILED ( 11, "Output file CLOSE failed"),
    OUTPUT_NEWLINE_FAILED ( 10, "Output newline failed"),
    OUTPUT_WRITE_FAILED (  9, "Output WRITE failed"),
    OUT_ENCOD_UNSUPPORTED (  8, "Output stream encoding not supported"),
    OUTPUT_SW_NOT_CREATED (  7, "Output stream writer not created"),
    OUTPUT_OPEN_FAILED (  6, "Output file OPEN failed"),
    INPUT_CLOSE_FAILED (  5, "Input file CLOSE failed"),
    INPUT_READ_FAILED (  4, "Input READ failed"),
    IN_ENCOD_UNSUPPORTED (  3, "Input stream encoding not supported"),
    INPUT_SR_NOT_CREATED (  2, "Input stream reader not created"),
    INPUT_OPEN_FAILED (  1, "Input file OPEN failed"),
    OK ( 0, "OK"),
    ILLEGAL_FILE_ID ( -1, "File ID < 0 or > max. value"),
    MISSING_FILE_NAME ( -2, "Missing file name"),
    OPENED_TWICE ( -3, "File already open"),
    FILE_NOT_OPEN ( -4, "File not open"),
    IN_MAX_FILES_DEFINED ( -5, "Max. no. of input files already defined"),
    ILLEGAL_IN_MAX_FILES ( -6, "Max. no. of input files <= 0"),
    OUT_MAX_FILES_DEFINED ( -7, "Max. no. of output files already defined"),
    ILLEGAL_OUT_MAX_FILES ( -8, "Max. no. of output files <= 0"),
    INPUT_REF_MISMATCH ( -9, "Input reference mismatch"),
    OUTPUT_REF_MISMATCH (-10, "Output reference mismatch"),
    INPUT_REF_MISSING (-11, "Input reference missing"),
    OUTPUT_REF_MISSING (-12, "Output reference missing");
}
```

```

private final int    code;
private final String message;

FS_Code(int code, String message)
{
    this.code    = code;
    this.message = message;
}

public int    getCode()    { return code; }

public String getMessage()
{
    if (this.code == OK.getCode())
        { return message; }
    else
        { return "++++ " + message; }
}
}

```

### Anmerkungen

Die Klasse `FS_Code` ordnet den numerischen Fehlercodes der File Services kurze Fehlertexte zu. Die positiven Fehlercodes sind für Exceptions vorgesehen. Sie werden – nach dem Schließen aller offenen Dateien – als `exitCode` an `System.exit()` übergeben und dürfen deshalb maximal den Wert 255 haben.

Die negativen Fehlercodes sind diversen Konsistenzfehlern zugeordnet und werden sowohl innerhalb der File Services als auch in der diese nutzenden Anwendung verwendet. Die Anwendung entscheidet darüber, ob dem Auftreten eines solchen Fehlers ein kontrollierter Laufabbruch – Shutdown – folgen soll, oder ob der Fehler lediglich protokolliert wird.

Eine kleine Besonderheit ist, dass `getMessage()` den Fehlertexten bis auf den OK-Fall – Code Null – vier Pluszeichen und ein Leerzeichen voranstellt. Diese optische Hervorhebung hat sich bewährt.

## 8.5 Klasse Trace

```

package FileService;

/**
 * This enumeration defines the switches for the trace statements.
 */
public enum Trace
{
    DETAIL    ( false, "Display detail data"),
    PATH      ( false, "Display path mnemonics & numbers");

    private final boolean traceFlag;
    private final String purpose;
}

```

```

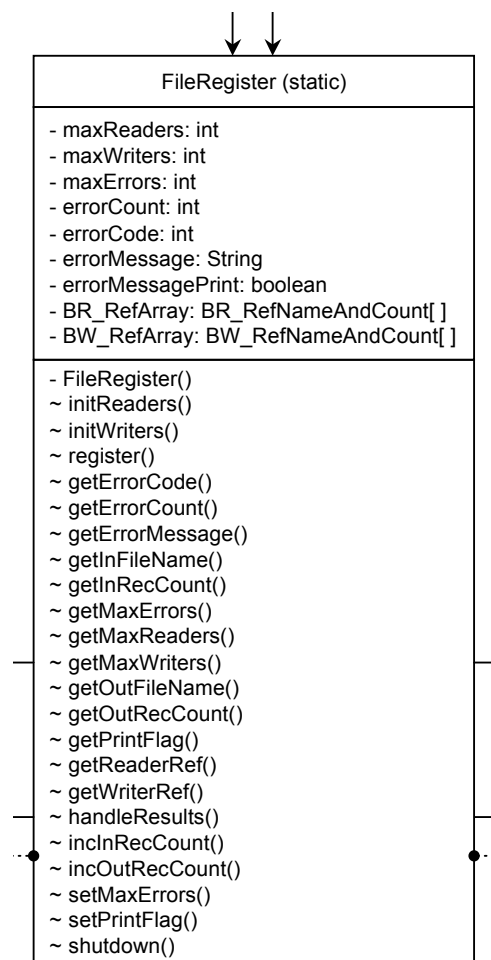
Trace(boolean traceFlag, String purpose)
{
    this.traceFlag = traceFlag;
    this.purpose     = purpose;
}

public boolean active()      { return traceFlag; }

public String getPurpose() { return purpose; }
}

```

## 8.6 Klasse FileRegister



```

package FileService;

import java.io.BufferedReader;
import java.io.BufferedWriter;

/**
 * The class FileRegister is the repository for the file input and output
 * references and for associated information.
 * It supports the classes InFileUtil and OutFileUtil by
 * - allocating the arrays, based on the "readFiles" / "writeFiles" parameters
 * - registering newly opened files and de-registering them on close
 * - checking the I/O requests against the repository data
 * - returning the correct BufferedReader / BufferedWriter references
 * - counting the read / write accesses (by delegation)
 * - returning a file name & record count for a given open file
 * - closing all open files on a shutdown request, then System.exit(<exit code>)
 * - returning error feedback values to the callers, if necessary.
 * The methods of FileRegister are private (constructor only) or packet private.
 */
class FileRegister
{
    private static int          maxReaders;
    private static int          maxWriters;
    private static int          maxErrors;
    private static int          errorCount;
    private static int          errorCode;
    private static String       errorMessage;
    private static boolean      errorMessagePrint;
    private static BR_RefNameAndCount[] BR_RefArray;
    private static BW_RefNameAndCount[] BW_RefArray;

    static
    {
        maxReaders      = 0;
        maxWriters      = 0;
        maxErrors       = -1;
        errorCount       = 0;
        errorCode        = FS_Code.OK.getCode();
        errorMessage     = FS_Code.OK.getMessage();
        errorMessagePrint = true;
    }
    /**
     * dummy constructor for objects of class FileRegister
     */
    private FileRegister() { }

```

### 8.6.1 initReaders()

```

/**
 * initReaders() can be used only once for allocating the BR_RefNameAndCount
 * array. The parameter readFiles must be positive.
 */
static int initReaders(int readFiles)
{
    if (maxReaders > 0)
    {
        return handleResults(FS_Code.IN_MAX_FILES_DEFINED.getCode(),
                             FS_Code.IN_MAX_FILES_DEFINED.getMessage());
    }

    if (readFiles <= 0)
    {
        return handleResults(FS_Code.ILLEGAL_IN_MAX_FILES.getCode(),
                             FS_Code.ILLEGAL_IN_MAX_FILES.getMessage());
    }

    maxReaders = readFiles;
    BR_RefArray = new BR_RefNameAndCount[readFiles];

    return handleResults(FS_Code.OK.getCode(),
                         FS_Code.OK.getMessage());
}

```

### 8.6.2 initWriters()

```

/**
 * initWriters() can be used only once for allocating the BW_RefNameAndCount
 * array. The parameter writeFiles must be positive.
 */
static int initWriters(int writeFiles)
{
    if (maxWriters > 0)
    {
        return handleResults(FS_Code.OUT_MAX_FILES_DEFINED.getCode(),
                             FS_Code.OUT_MAX_FILES_DEFINED.getMessage());
    }

    if (writeFiles <= 0)
    {
        return handleResults(FS_Code.ILLEGAL_OUT_MAX_FILES.getCode(),
                             FS_Code.ILLEGAL_OUT_MAX_FILES.getMessage());
    }
}

```

```

        maxWriters = writeFiles;
        BW_RefArray = new BW_RefNameAndCount[writeFiles];

        return handleResults(FS_Code.OK.getCode(),
                             FS_Code.OK.getMessage());
    }

```

### 8.6.3 register()

```

/**
 * This register() method stores the attributes of an input file.
 */
static int register(int fileID,
                    BufferedReader readerRef,
                    String inFileName)
{
    String errorMsg;

    if (fileID < 0 || fileID >= maxReaders)
    {
        errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
            " (" + (maxReaders-1) +
            "): >" + fileID + "< (Register Input)";

        return handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
                             errorMsg);
    }

    if (readerRef == null)
    {
        errorMsg = FS_Code.INPUT_REF_MISSING.getMessage() +
            " for file ID >" + fileID + "< (Register Input)";

        return handleResults(FS_Code.INPUT_REF_MISSING.getCode(),
                             errorMsg);
    }

    if (inFileName == null)
    {
        errorMsg = FS_Code.MISSING_FILE_NAME.getMessage() +
            " for file ID >" + fileID + "< (Register Input)";

        return handleResults(FS_Code.MISSING_FILE_NAME.getCode(),
                             errorMsg);
    }

    BR_RefArray[fileID] = new BR_RefNameAndCount(readerRef, inFileName);
}

```

```

        return handleResults(FS_Code.OK.getCode(),
                               FS_Code.OK.getMessage());
    }

    /**
     * This register() method removes the attributes of an input file.
     */
    static int register(int          fileID,
                        BufferedReader readerRef)
    {
        String errorMsg;

        if (fileID < 0 || fileID >= maxReaders)
        {
            errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
                " (" + (maxReaders-1) +
                "): >" + fileID + "< (De-Register Input)";

            return handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
                                errorMsg);
        }

        if (BR_RefArray[fileID].getReaderRef() != readerRef)
        {
            errorMsg = FS_Code.INPUT_REF_MISMATCH.getMessage() +
                " for file ID >" + fileID + "< (De-Register Input)";

            return handleResults(FS_Code.INPUT_REF_MISMATCH.getCode(),
                                errorMsg);
        }

        BR_RefArray[fileID] = null;

        return handleResults(FS_Code.OK.getCode(),
                               FS_Code.OK.getMessage());
    }

    /**
     * This register() method stores the attributes of an output file.
     */
    static int register(int          fileID,
                        BufferedWriter writerRef,
                        String          outFileName)
    {
        String errorMsg;

```

```

    if (fileID < 0 || fileID >= maxWriters)
    {
        errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
            " (" + (maxWriters-1) +
            "): >" + fileID + "< (Register Output)";

        return handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
            errorMsg);
    }

    if (writerRef == null)
    {
        errorMsg = FS_Code.OUTPUT_REF_MISSING.getMessage() +
            " for file ID >" + fileID + "< (Register Output)";

        return handleResults(FS_Code.OUTPUT_REF_MISSING.getCode(),
            errorMsg);
    }

    if (outFileName == null)
    {
        errorMsg = FS_Code.MISSING_FILE_NAME.getMessage() +
            " for file ID >" + fileID + "< (Register Output)";

        return handleResults(FS_Code.MISSING_FILE_NAME.getCode(),
            errorMsg);
    }

    BW_RefArray[fileID] = new BW_RefNameAndCount(writerRef, outFileName);

    return handleResults(FS_Code.OK.getCode(),
        FS_Code.OK.getMessage());
}

/**
 * This register() method removes the attributes of an output file.
 */
static int register(int fileID,
    BufferedWriter writerRef)
{
    String errorMsg;

    if (fileID < 0 || fileID >= maxWriters)
    {
        errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
            " (" + (maxWriters-1) +
            "): >" + fileID + "< (De-Register Output)";
    }

```

```

        return handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
                             errorMsg);
    }

    if (BW_RefArray[fileID].getWriterRef() != writerRef)
    {
        errorMsg = FS_Code.OUTPUT_REF_MISMATCH.getMessage() +
            " for file ID >" + fileID + "< (De-Register Output)";

        return handleResults(FS_Code.OUTPUT_REF_MISMATCH.getCode(),
                             errorMsg);
    }

    BW_RefArray[fileID] = null;

    return handleResults(FS_Code.OK.getCode(),
                        FS_Code.OK.getMessage());
}

```

#### 8.6.4 getXXX() methods

```

static int    getErrorCode()    { return errorCode; }

static int    getErrorCount()   { return errorCount; }

static String getErrorMessage() { return errorMessage; }

/**
 * getInFileName() obtains the input file name for a given file ID.
 */
static String getInFileName(int fileID)
{
    String fileName = null;

    if (fileID >= 0          &&
        fileID <  maxReaders &&
        BR_RefArray[fileID] != null)
        { fileName = BR_RefArray[fileID].getFileName(); }

    return fileName;
}

/**
 * getInRecCount() obtains the read count for a given file ID.
 */
static long getInRecCount(int fileID)

```

```

{
    long crtCount = -1;

    if (fileID < 0 || fileID >= maxReaders)
    { }
    else
        if (BR_RefArray[fileID] != null)
            { crtCount = BR_RefArray[fileID].getRecCount(); }

    return crtCount;
}

static int    getMaxErrors() { return maxErrors; }

static int    getMaxReaders() { return maxReaders; }

static int    getMaxWriters() { return maxWriters; }

/**
 * getOutFileName() obtains the output file name for a given file ID.
 */
static String getOutFileName(int fileID)
{
    String fileName = null;

    if (fileID >= 0      &&
        fileID <  maxWriters &&
        BW_RefArray[fileID] != null)
        { fileName = BW_RefArray[fileID].getFileName(); }

    return fileName;
}

/**
 * getOutRecCount() obtains the write count for a given file ID.
 */
static long getOutRecCount(int fileID)
{
    long crtCount = -1;

    if (fileID < 0 || fileID >= maxWriters)
    { }
    else
        if (BW_RefArray[fileID] != null)
            { crtCount = BW_RefArray[fileID].getRecCount(); }

    return crtCount;
}

```

```

    }

    static boolean getPrintFlag() { return errorMessagePrint; }

    /**
     * getReaderRef() obtains a BufferedReader reference.
     */
    static BufferedReader getReaderRef(int fileID)
    {
        if (BR_RefArray[fileID] == null)
        { return null; }
        else
        { return BR_RefArray[fileID].getReaderRef(); }
    }

    /**
     * getWriterRef() obtains a BufferedWriter reference.
     */
    static BufferedWriter getWriterRef(int fileID)
    {
        if (BW_RefArray[fileID] == null)
        { return null; }
        else
        { return BW_RefArray[fileID].getWriterRef(); }
    }

```

### 8.6.5 handleResults()

```

static int handleResults(int code, String message)
{
    errorCode    = code;
    errorMessage = message;

    if (code != FS_Code.OK.getCode())
    {
        errorCount++;

        if (errorMessagePrint)
        { System.out.println(message); }
    }

    return code;
}

```

### 8.6.6 incInRecCount()

```
/**
 * incInRecCount() increments the read count for a given file ID.
 */
static int incInRecCount(int fileID)
{
    if (fileID < 0 || fileID >= maxReaders)
        { return FS_Code.ILLEGAL_FILE_ID.getCode(); }

    if (BR_RefArray[fileID] == null)
        { return FS_Code.FILE_NOT_OPEN.getCode(); }
    else
        { BR_RefArray[fileID].incRecCount(); }

    return FS_Code.OK.getCode();
}
```

8

### 8.6.7 incOutRecCount()

```
/**
 * incOutRecCount() increments the write count for a given file ID.
 */
static int incOutRecCount(int fileID)
{
    if (fileID < 0 || fileID >= maxWriters)
        { return FS_Code.ILLEGAL_FILE_ID.getCode(); }

    if (BW_RefArray[fileID] == null)
        { return FS_Code.FILE_NOT_OPEN.getCode(); }
    else
        { BW_RefArray[fileID].incRecCount(); }

    return FS_Code.OK.getCode();
}
```

### 8.6.8 setXXX() methods

```
static void setMaxErrors(int errorLimit) { maxErrors = errorLimit; }

static void setPrintFlag(boolean errMsgPrint)
{ errorMessagePrint = errMsgPrint; }
```

## 8.6.9 shutdown()

```

/**
 * shutdown() closes all registered input / output files
 * except the one with a failed open / close request.
 * Then it stops the program with the specified exit code.
 */
static void shutdown(int    fileID,
                    char    in_out,
                    char    accessMode,
                    int     exitCode,
                    String  shutdownReason)
{
    // pause the thread: let the stack dump complete
    try {
        Thread.sleep(100);
    }
    catch (Exception ex) { }

    if (Trace.PATH.active()) { System.out.println("FRSD01"); }

    System.out.println("++++ Shutdown requested with code " + exitCode);

    if (shutdownReason != null)
    {
        if (Trace.PATH.active()) { System.out.println("FRSD02"); }

        System.out.println(shutdownReason);
    }

    for (int i = 0; i < maxReaders; i++)
    {
        if (Trace.PATH.active()) { System.out.println("FRSD03"); }

        if (BR_RefArray[i] != null)
        {
            if (Trace.PATH.active()) { System.out.println("FRSD04"); }

            if (in_out == 'I' &&
                accessMode != 'R' &&
                i == fileID)
            { if (Trace.PATH.active()) { System.out.println("FRSD05"); } }
            else
            {
                if (Trace.PATH.active()) { System.out.println("FRSD06"); }

                try {

```

```

        System.out.println("Trying to close input file " +
                           BR_RefArray[i].getFileName());

        System.out.println("Records read = " +
                           BR_RefArray[i].getRecCount());

        BR_RefArray[i].getReaderRef().close();

        System.out.println("Input file emergency CLOSE successful");
    }
    catch(Exception em) {
        System.out.println("Input file emergency CLOSE failed");
    }
}

if (Trace.PATH.active()) { System.out.println("FRSD07"); }

BR_RefArray[i] = null;
}
}

for (int i = 0; i < maxWriters; i++)
{
    if (Trace.PATH.active()) { System.out.println("FRSD08"); }

    if (BW_RefArray[i] != null)
    {
        if (Trace.PATH.active()) { System.out.println("FRSD09"); }

        if (in_out      == 'O' &&
            accessMode != 'W' &&
            i == fileID)
        { if (Trace.PATH.active()) { System.out.println("FRSD10"); } }
        else
        {
            if (Trace.PATH.active()) { System.out.println("FRSD11"); }

            try {
                System.out.println("Trying to close output file " +
                                   BW_RefArray[i].getFileName());

                System.out.println("Records written = " +
                                   BW_RefArray[i].getRecCount());

                BW_RefArray[i].getWriterRef().close();

                System.out.println("Output file emergency CLOSE successful");
            }
            catch (Exception em) {
                System.out.println("Output file emergency CLOSE failed");
            }
        }
    }
}

```

```

        }
        catch(Exception em) {
            System.out.println("Output file emergency CLOSE failed");
        }
    }
    if (Trace.PATH.active()) { System.out.println("FRSD12"); }

    BW_RefArray[i] = null;
}

if (Trace.PATH.active()) { System.out.println("FRSD13"); }

System.exit(exitCode);
}
}

```

### Anmerkungen

Die Klasse `FileRegister` hat vielfältige Aufgaben. Sie

- dient als Repository für Dateinamen, Zugriffszähler und Zugriffsobjekt-Referenzen, getrennt nach Eingabe und Ausgabe,
- legt die dafür erforderlichen Arrays an und verwaltet sie,
- speichert Fehlerdaten und aktualisiert diese nach allen dafür relevanten Aktionen,
- führt das Fehlerlimit und den Fehlerausgabe-Schalter,
- bietet 13 Getter-Methoden an, die `InFileUtil` und `OutFileUtil` unterstützen, oder die Anfragen aus der Anwendung zu beantworten ermöglichen,
- leitet Zugriffszähler-Inkrementierungsaufrufe durch, und
- vollzieht a) auf Anforderung, b) bei Überschreiten des Fehlerlimits oder c) nach schweren internen Fehlern einen kontrollierten Laufabbruch mit Schließen aller offenen Dateien und Aufruf von `System.exit()`.

Im Gegensatz zu den Front End-Klassen `InFileUtil` und `OutFileUtil` besitzt die Klasse `FileRegister` eigene Variable. Diese sind sämtlich `static` und haben teilweise eigene Initialisierungen, die mit dem Schlüsselwort `static` eingeleitet werden. Diese Defaults lauten: a) unbegrenzte Fehlerzahl (negative Zahl = kein Limit), b) Zuweisung von Null als aktueller Fehlercode und 'OK' als aktueller Fehlertext, sowie c) Protokollierung von Fehlern durch die File Services = eingeschaltet.

Die Methoden von `FileRegister` sind bis auf `initxxx()` und die vier `register()`-Varianten alphabetisch aufsteigend geordnet, um die Übersicht zu wahren. Daher werden sie nachfolgend thematisch gruppiert besprochen, soweit das als sinnvoll erscheint.

**initReaders()**

**initWriters()**

**getMaxReaders()**

**getMaxWriters()**

Das Anwendungsprogramm legt die maximalen Anzahlen gleichzeitig geöffneter Ein- und Ausgabedateien unabhängig voneinander – normalerweise zu Laufbeginn – unveränderlich fest, indem es `setupInFiles()` – `InFileUtil` – und `setupOutFiles()` – `OutFileUtil` – aufruft. Diese Aufrufe werden ungeprüft an `initReaders()` / `initWriters()` weitergegeben. Dort legen diese `FileRegister`-Methoden den jeweils gewünschten Array an und merken sich die ihm zugeordnete Anzahl der Einträge, die natürlich größer als Null sein muss. Weitere Aufrufe für dieselben Arrays werden – mit Fehlerrückmeldung – abgelehnt.

Solange kein Setup für Eingabe erfolgt ist, können keine Eingabedateien geöffnet werden; dies gilt analog für die Ausgabe. Somit kann ein Programm zunächst ganz ohne Dateien arbeiten und später beispielsweise nur Eingabedateien oder nur Ausgabedateien verwenden.

Wenn Dateien in mehreren Rollen verwendet werden, müssen dementsprechend auch mehr Einträge verlangt werden. Verarbeitet das Programm dagegen etwa eine Serie von Bilddateien zwecks automatischer Skalierung und Ausgabe der skalierten Bilder als neue Serie, so ist dafür nur je eine Eingabe und eine Ausgabe vorzusehen.

Die Anzahlen können mit den beiden `getMaxxxx()`-Methoden abgefragt werden, zum Beispiel um sicherzustellen, dass die Datei-Identifikatoren in einer Enumeration die dadurch definierten Grenzen einhalten, also minimal den Wert Null und maximal die jeweilige Anzahl minus 1 aufweisen.

### **register()**

`FileRegister` besitzt vier `register()`-Methoden, die sich durch ihre Signaturen unterscheiden und die seitens `InFileUtil` / `OutFileUtil` bei erfolgreichem Input-Open, Input-Close, Output-Open oder Output-Close aufgerufen werden. Diese Methoden verwalten die Einträge in den beiden Arrays.

Nach einem Open werden der Name der soeben geöffneten Datei und die neue `BufferedReader`-/`BufferedWriter`-Referenz an den jeweiligen Array-Eintrags-Konstruktor übergeben. Ein Close hat zur Folge, dass der davon betroffene Eintrag dem Garbage Collector übergeben wird. Somit können unter derselben Datei-Identifikation nacheinander mehrere Dateien oder dieselbe Datei mehrfach verwendet werden, falls gewünscht.

Diese Array-Operationen müssen absolut korrekt ablaufen. Daher sind zahlreiche Parameterprüfungen vorgesehen.

**getInFileName()  
getInRecCount()  
getOutFileName()  
getOutRecCount()  
getReaderRef()  
getWriterRef()  
incInRecCount()  
incOutRecCount()**

sind Methoden, die sich mit den Array-Einträgen befassen, um Informationen zu beschaffen und die Zugriffszähler zu führen. Die ersten vier Getter werden von `InFileUtil` und `OutFileUtil` verwendet, um externe Anfragen zu beantworten, aber auch für eigene Zwecke. Die beiden `getxxxRef()`-Methoden liefern die von `InFileUtil` und `OutFileUtil` benötigten Zugriffsobjekt-Referenzen für Read / Write / Close zurück.

**getErrorCode()**  
**getErrorCount()**  
**getErrorMessage()**  
**getMaxErrors()**  
**setMaxErrors()**

werden von `InFileUtil` und `OutFileUtil` sowohl publiziert als auch dazu verwendet, das Überschreiten des Fehlerlimits festzustellen und einen kontrollierten Laufabbruch einzuleiten. Für Letzeres sind `getErrorCount()` und `getMaxErrors()` zuständig. Im `FileRegister` sind vier Klassenvariable für diese Zwecke vorhanden. Anwendungsprogramme können sich jederzeit die aktuellste Fehlermeldung holen und selbst für Ausgabezwecke verwenden.

**getPrintFlag()**  
**setPrintFlag()**

fragen den aktuellen Status der Protokollierung von Fehlern durch die File Services ab oder setzen diesen Status. Bei ausgeschalteter Protokollierung werden zwar die Fehlercodes und -texte intern abgelegt, aber es kommt zu keiner Protokollausgabe. Das Anwendungsprogramm muss in diesem Fall dafür sorgen, dass relevante Fehlermeldungen in geeigneter Weise sichtbar gemacht werden.

Shutdown-Protokolle werden jedoch in jedem Fall ausgegeben, da dann das Anwendungsprogramm die Kontrolle nicht zurückbekommt.

**handleResults()**

speichert die übergebenen Fehlerdaten in den Klassenvariablen und erhöht – ausser im OK-Fall – den Fehlerzähler. Falls die Protokollierung von Fehlern durch die File Services eingeschaltet ist, gibt `handleResults()` zudem den Fehlertext auf das Protokollmedium aus.

Das Fortschalten des Fehlerzählers kann dazu führen, dass das Fehlerlimit überschritten wird. In diesem Fall warten die File Services den nächsten Aufruf seitens der Anwendung ab, der irgendeine Veränderung bewirken würde, also eine File-Operation oder einen `setupXXXFiles()`-Aufruf. Erst dann wird der Shutdown eingeleitet. Deshalb kann die Anwendung den betreffenden Fehler selbst behandeln beziehungsweise protokollieren.

**shutdown()**

wartet zu Beginn kurze Zeit, um den von `InFileUtil` oder `OutFileUtil` gegebenenfalls angeforderten Stack Trace Dump vollständig ausgeben zu lassen. Das geschieht anscheinend in einem anderen Thread. Ohne dieses Warten vermischen sich die Ausgaben von `shutdown()` auf irritierende Weise mit dem Dump.

Danach meldet `shutdown()` den Aufruf und den `exitCode` im Protokoll. Ein eventuell übergebener Zusatztext `shutdownReason` wird ebenfalls protokolliert. In zwei aufeinanderfolgenden Schleifen schließt `shutdown()` alle offenen Dateien, protokolliert dabei deren Namen und Zugriffszählerstände, und meldet das erfolgreiche Schließen. Dabei wird die verursachende Datei ausgeklammert, sofern eine Exception bei Open oder Close auftrat, weil dann ein erneuter Close-Versuch nicht sinnvoll ist.

Falls bei diesen Close-Aufrufen Exceptions auftreten, wird jeweils nur eine Mitteilung ausgegeben (... file emergency CLOSE failed).

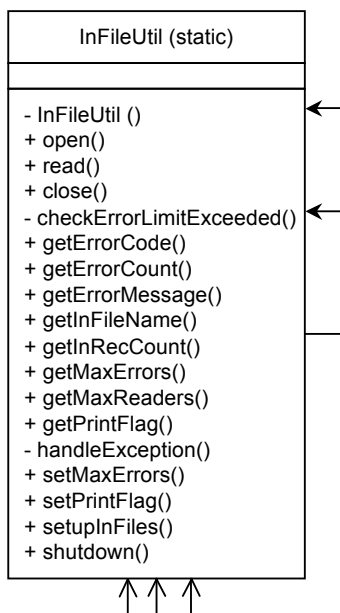
### Schlussbemerkung zu FileRegister

Beim Betrachten des Codes fällt rasch auf, dass ein- und ausgabebezogene Operationen respektive Methoden nicht als polymorphe Varianten gehandhabt, sondern jeweils ausprogrammiert wurden. Aus der Sicht einer möglichst durchgreifenden „Objektierung“ erscheint das als unbefriedigend. Theoretisch wäre es ja denkbar, diese Varianten einer gemeinsamen Klasse zuzuordnen, welche die arraybezogenen Aspekte unabhängig von der Transferrichtung abdeckt. Je ein Objekt dieser Klasse könnte in `InFileUtil` / `OutFileUtil` erzeugt und registriert werden. Für die Klasse `FileRegister` blieben dann nur die Fehlerinformations-Methoden, die Fehlerlimit-Überwachung, die Protokollierungs-Schalter-Methoden und die Shutdown-Logik.

Eine solche Aufgabenverteilung wäre derer in `EvaluateSportsDS` ähnlich, wo `Contest_LL` die link-listen-bezogenen Aufgaben wahrnimmt. Allerdings führte der Versuch, diese Analogie zu nutzen, zu keinem brauchbaren Ergebnis. Dieser Code war nicht elegant, sondern erzwang zusätzliche Parameter und Abfragen für die Erledigung derselben Aufgaben. Die interne Verflechtung der Methoden-Aufrufe nahm wegen der beim `FileRegister` verbleibenden zentralen Aufgaben sogar zu. Es wurde also keine überzeugende Vereinfachung erreicht.

Daher blieb es bei dem oben aufgelisteten Code. Klarheit, Einfachheit und Sicherheit gehen vor. Vor allem der Sicherheitsaspekt prägt die meisten Methoden durch die darin enthaltenen Parameterprüfungen. Damit wird ein zweites Sicherheitsnetz unterhalb der Front End-Klassen `InFileUtil` und `OutFileUtil` eingezogen, das zuverlässig eine Fehlerausbreitung verhindert – auch wenn etliche Prüfungen als redundant erscheinen.

## 8.7 Klasse InFileUtil



```
package FileService;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;

/**
 * The static class InFileUtil wraps all input file operations.
 */

public final class InFileUtil
{
    /**
     * dummy constructor for objects of class InFileUtil
     */
    private InFileUtil() { }
```

### 8.7.1 open()

```
/**
 * open() checks first if the error limit is exceeded (==> shut down);
 * otherwise it verifies the fileID plus the fileName superficially. Then it
 * asks the FileRegister whether a BufferedReader entry exists for the fileID.
 * If the file is not already registered, a file input stream is opened,
 * using the input file name. Otherwise the request is rejected.
 * After successfully creating the input stream, an input stream reader
 * is created with or without a certain encoding:
 * - A null encoding string reference means: no encoding.
 * - A non-null encoding string reference means: use it in the call.
 * Finally, a buffered reader object is created and its reference plus
 * other attributes are stored in the associated FileRegister entry.
 */
public static int open(int    fileID,
                      String fileName,
                      String encoding)
{
    FileInputStream    inputStream = null;
    InputStreamReader streamReader = null;
    String              errorMsg;

    if (Trace.PATH.active()) { System.out.println("IFOP01"); }

    checkErrorLimitExceeded();

    if (Trace.PATH.active()) { System.out.println("IFOP02"); }

    if (fileID < 0 || fileID >= FileRegister.getMaxReaders())
```

```

{
    if (Trace.PATH.active()) { System.out.println("IFOP03"); }

    errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
        " (" + (FileRegister.getMaxReaders()-1) +
        "): >" + fileID + "< (Input OPEN)";

    return FileRegister.handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
        errorMsg);
}

if (Trace.PATH.active()) { System.out.println("IFOP04"); }

if (fileName == null)
{
    if (Trace.PATH.active()) { System.out.println("IFOP05"); }

    errorMsg = FS_Code.MISSING_FILE_NAME.getMessage() +
        ": >" + fileID + "< (Input OPEN)";

    return FileRegister.handleResults(FS_Code.MISSING_FILE_NAME.getCode(),
        errorMsg);
}

if (Trace.PATH.active()) { System.out.println("IFOP06"); }

BufferedReader reader = FileRegister.getReaderRef(fileID);

if (reader != null)
{
    if (Trace.PATH.active()) { System.out.println("IFOP07"); }

    errorMsg = FS_Code.OPENED_TWICE.getMessage() +
        " for ID >" + fileID + "< (Input OPEN)";

    return FileRegister.handleResults(FS_Code.OPENED_TWICE.getCode(),
        errorMsg);
}

if (Trace.PATH.active()) { System.out.println("IFOP08"); }

try {
    inputStream = new FileInputStream(fileName);
}
catch(Exception ex) {
    handleException(fileID, null, 'O', ex,
        FS_Code.INPUT_OPEN_FAILED.getCode(),

```

```

        FS_Code.INPUT_OPEN_FAILED.getMessage());
    }

    if (Trace.PATH.active()) { System.out.println("IFOP09"); }

    if (encoding == null)
    {
        if (Trace.PATH.active()) { System.out.println("IFOP10"); }

        try {
            streamReader = new InputStreamReader(inputStream);
        }
        catch(Exception ex) {
            handleException(fileID, fileName, 'O', ex,
                FS_Code.INPUT_SR_NOT_CREATED.getCode(),
                FS_Code.INPUT_SR_NOT_CREATED.getMessage());
        }
    }
    else
    {
        if (Trace.PATH.active()) { System.out.println("IFOP11"); }

        try {
            streamReader = new InputStreamReader(inputStream, encoding);
        }
        catch(Exception ex) {
            handleException(fileID, fileName, 'O', ex,
                FS_Code.IN_ENCOD_UNSUPPORTED.getCode(),
                FS_Code.IN_ENCOD_UNSUPPORTED.getMessage());
        }
    }

    if (Trace.PATH.active()) { System.out.println("IFOP12"); }

    reader = new BufferedReader(streamReader);

    if (FileRegister.register(fileID, reader, fileName) != FS_Code.OK.getCode())
    {
        shutdown(FS_Code.INTERNAL_FATAL_ERROR.getCode(),
            FS_Code.INTERNAL_FATAL_ERROR.getMessage());
    }

    if (Trace.PATH.active()) { System.out.println("IFOP13"); }

    return FileRegister.handleResults(FS_Code.OK.getCode(),
        FS_Code.OK.getMessage());
}

```

## 8.7.2 read()

```

/**
 * read() reads an input record and returns it as a String.
 */
public static String read(int fileID)
{
    String inputString = null;
    String errorMsg;

    checkErrorLimitExceeded();

    if (fileID < 0 || fileID >= FileRegister.getMaxReaders())
    {
        errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
            " (" + (FileRegister.getMaxReaders()-1) +
            "): >" + fileID + "< (READ)";

        FileRegister.handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
            errorMsg);

        return null;
    }

    BufferedReader reader = FileRegister.getReaderRef(fileID);

    if (reader == null)
    {
        errorMsg = FS_Code.FILE_NOT_OPEN.getMessage() +
            " with ID >" + fileID + "< (READ)";

        FileRegister.handleResults(FS_Code.FILE_NOT_OPEN.getCode(),
            errorMsg);

        return null;
    }

    try {
        inputString = reader.readLine();
    }
    catch(Exception ex) {
        handleException(fileID, null, 'R', ex,
            FS_Code.INPUT_READ_FAILED.getCode(),
            FS_Code.INPUT_READ_FAILED.getMessage());
    }

    // System.out.println("\t inputString = " + inputString);

```

```

// increment read count if EOF is not reached
if (inputString != null &&
    FileRegister.incInRecCount(fileID) != FS_Code.OK.getCode())
{
    shutdown(FS_Code.INTERNAL_FATAL_ERROR.getCode(),
            FS_Code.INTERNAL_FATAL_ERROR.getMessage());
}

FileRegister.handleResults(FS_Code.OK.getCode(),
    FS_Code.OK.getMessage());

return inputString;
}

```

### 8.7.3 close()

```

/**
 * close() closes the input file.
 */
public static int close(int fileID)
{
    String errorMsg;

    checkErrorLimitExceeded();

    if (fileID < 0 || fileID >= FileRegister.getMaxReaders())
    {
        errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
            " (" + (FileRegister.getMaxReaders()-1) +
            "): >" + fileID + "< (Input CLOSE)";

        return FileRegister.handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
            errorMsg);
    }

    BufferedReader reader = FileRegister.getReaderRef(fileID);

    if (reader == null)
    {
        errorMsg = FS_Code.FILE_NOT_OPEN.getMessage() +
            " with ID >" + fileID + "< (Input CLOSE)";

        return FileRegister.handleResults(FS_Code.FILE_NOT_OPEN.getCode(),
            errorMsg);
    }
}

```

```

    try {
        reader.close();
    }
    catch(Exception ex) {
        handleException(fileID, null, 'C', ex,
            FS_Code.INPUT_CLOSE_FAILED.getCode(),
            FS_Code.INPUT_CLOSE_FAILED.getMessage());
    }

    if (FileRegister.register(fileID, reader) != FS_Code.OK.getCode())
    {
        shutdown(FS_Code.INTERNAL_FATAL_ERROR.getCode(),
            FS_Code.INTERNAL_FATAL_ERROR.getMessage());
    }

    return FileRegister.handleResults(FS_Code.OK.getCode(),
        FS_Code.OK.getMessage());
}

```

### 8.7.4 checkErrorLimitExceeded()

```

/**
 * checkErrorLimitExceeded() obtains the error limit & count values.
 * If the error limit is non-negative and the error count exceeds it,
 * the shutdown sequence is activated.
 */
private static void checkErrorLimitExceeded()
{
    int maxErrors = FileRegister.getMaxErrors();

    if (maxErrors >= 0 && FileRegister.getErrorCount() > maxErrors)
    {
        shutdown(FS_Code.ERROR_LIMIT_EXCEEDED.getCode(),
            FS_Code.ERROR_LIMIT_EXCEEDED.getMessage() +
            ": >" + maxErrors + "<");
    }
}

```

### 8.7.5 getXXX() methods

```

/**
 * The getXXX() methods encapsulate the corresponding FileRegister methods.
 */

public static int getErrorCode()
{ return FileRegister.getErrorCode(); }

```

```

public static int      getErrorCount()
{ return FileRegister.getErrorCount(); }

public static String   getErrorMessage()
{ return FileRegister.getErrorMessage(); }

public static String   getInFileName(int fileID)
{ return FileRegister.getInFileName(fileID); }

public static long     getInRecCount(int fileID)
{ return FileRegister.getInRecCount(fileID); }

public static int      getMaxErrors()
{ return FileRegister.getMaxErrors(); }

public static int      getMaxReaders()
{ return FileRegister.getMaxReaders(); }

public static boolean  getPrintFlag()
{ return FileRegister.getPrintFlag(); }

```

### 8.7.6 **handleException()**

```

/**
 * handleException() combines the terminating actions.
 */
private static void handleException(int      fileID,
                                     String    fileName,
                                     char      accessMode,
                                     Exception  excp,
                                     int       exitCode,
                                     String     errorMsg)
{
    String inFileName;

    if (Trace.PATH.active()) { System.out.println("IFHE01"); }

    System.out.println(errorMsg);

    if (fileName == null)
    {
        if (Trace.PATH.active()) { System.out.println("IFHE02"); }

        inFileName = FileRegister.getInFileName(fileID);
    }
    else
    {

```

```

        if (Trace.PATH.active()) { System.out.println("IFHE03"); }

        inFileName = fileName;
    }

    if (inFileName != null)
    {
        if (Trace.PATH.active()) { System.out.println("IFHE04"); }

        System.out.println("Input file name = " + inFileName);

        long crtCount = FileRegister.getInRecCount(fileID);

        if (crtCount >= 0)
            { System.out.println("Records read = " + crtCount); }
    }

    if (Trace.PATH.active()) { System.out.println("IFHE05"); }

    // pause the thread: let the print output complete
    try {
        Thread.sleep(100);
    }
    catch (Exception ex) { }

    excp.printStackTrace();

    FileRegister.shutdown(fileID, 'I', accessMode, exitCode, null);
}

```

### 8.7.7 setXXX() methods

```

/**
 * The setXXX() and shutdown() methods encapsulate the corresponding
 * FileRegister methods.
 */
public static void setMaxErrors(int errorLimit)
{ FileRegister.setMaxErrors(errorLimit); }

public static void setPrintFlag(boolean errMsgPrint)
{ FileRegister.setPrintFlag(errMsgPrint); }

public static int setupInFiles(int readFiles)
{
    checkErrorLimitExceeded();

    return FileRegister.initReaders(readFiles);
}

```

```
}
```

### 8.7.8 shutdown()

```
public static void shutdown(int    exitCode,
                             String shutdownReason)
{ FileRegister.shutdown(-1, ' ', ' ', exitCode, shutdownReason); }
}
```

#### Anmerkungen

Die Klasse `InFileUtil` besteht hauptsächlich aus den Methoden `open()`, `read()`, `close()`, `checkErrorLimitExceeded()` und `handleException()`. Alle anderen `InFileUtil`-Methoden rufen lediglich die ihnen korrespondierenden `FileRegister`-Methoden auf. `setupInFiles()` prüft zusätzlich, ob das Fehlerlimit überschritten ist. Daher werden nachfolgend nur die fünf anfangs genannten Methoden kommentiert.

#### `open()`

Eingangs prüft `open()` zuerst, ob das Fehlerlimit überschritten ist. Falls nein, findet eine oberflächliche Prüfung der beiden ersten Parameter statt. Nach bestandener Fehlerlimit- und Parameterprüfung versucht `open()`, sich ein `BufferedReader`-Zugriffsobjekt vom `FileRegister` zu holen. Falls das gelingt, ist für die gegebene `fileID` bereits eine Datei – im Allgemeinen dieselbe – geöffnet und der Aufruf wird zurückgewiesen.

Andernfalls ist die `fileID` noch – im Array – unbelegt. Zunächst öffnet `open()` einen `FileInputStream`. Dabei wird unter anderem der Dateiname vom System geprüft und die Verbindung zur Datei hergestellt. Schlägt dies fehl, tritt also eine Exception ein, so bricht `open()` den Lauf über `handleException()` kontrolliert ab. Auch alle anderen Exceptions, die in `InFileUtil` auftreten können, enden so.

Nach erfolgreichem Öffnen erzeugt `open()` ein `InputStreamReader`-Objekt, das entweder kein Encoding vornimmt (also keine spezielle Zeichenkonvertierung beim Einlesen und Umsetzen in Unicode), oder den als Parameter vom Anwendungsprogramm übergebenen Encoding-String verwendet. In dieser Buchreihe ist das stets "ISO-8859-1", weil sich damit alle hier relevanten Sonderzeichen inklusive diverser diakritischer Zeichen aus Sprachen darstellen lassen, die auf der lateinischen Schrift basieren.

Der letzte Schritt, um ein für das Lesen verwendbares Zugriffsobjekt zu beschaffen, besteht im Erzeugen eines `BufferedReader`-Objekts, dessen Referenz direkt danach zusammen mit dem Dateinamen an die dafür vorgesehene `register()`-Methode übergeben wird. Der Registrierungsschritt muss korrekt ablaufen, weil sonst ein schwerer interner Fehler vorliegt, der zum Shutdown führt.

#### `read()`

setzt einen erfolgreichen `open()`-Vorgang voraus. Nach Fehlerlimit- und `fileID`-Prüfung versucht `read()` daher, eine gültige Lese-Zugriffsobjekt-Referenz von `FileRegister` zu holen. Wenn das fehlschlägt, ist die Datei – noch – nicht geöffnet und der Aufruf wird zurückgewiesen. Andernfalls findet mit der rückgemeldeten Referenz ein Leseversuch statt, dessen Fehlschlag einen Shutdown auslöst.

Nach erfolgreichem Lesen sind entweder Daten vorhanden oder das Dateiende (EOF) ist erreicht, was seitens `readLine()` mit `null` signalisiert wird. Nur im ersten Fall wird der Zugriffszähler erhöht, was unbedingt klappen muss. Abschliessend meldet `read()` den gelesenen String zurück.

Die Rückmeldung kann also aus zwei Gründen `null` sein: a) es trat ein – abgefangener – Fehler auf oder b) EOF liegt vor. Daher sollte das Anwendungsprogramm nach einer `null`-Rückmeldung mit dem Aufruf `InFileUtil.getErrorCode()` feststellen, ob EOF vorliegt, und den Lauf sonst gegebenenfalls abbrechen.

### **close()**

Auch `close()` benötigt nach bestandener Fehlerlimit-Prüfung eine gültige `fileID` und eine dadurch bezeichnete Lese-Zugriffsobjekt-Referenz. Diese ermöglicht, die betreffende Datei zu schließen. Das wird nahezu immer klappen; andernfalls folgt ein Shutdown.

Anschließend muss noch der bislang verwendete File-Register-Eintrag gelöscht werden. Gelingt das nicht, liegt ein schwerer interner Fehler vor, der zum Shutdown führt.

Wenn das Anwendungsprogramm den Namen einer Datei und ihren Zugriffszähler-Wert bei den File Services in Erfahrung bringen will, muss es dies spätestens unmittelbar vor dem `close()`-Aufruf tun, denn danach sind diese Angaben nicht mehr verfügbar.

### **checkErrorLimitExceeded()**

holt sich das aktuelle Fehlerlimit aus `FileRegister` und prüft, ob es positiv und – nun – überschritten ist. Trifft beides zu, leitet `checkErrorLimitExceeded()` einen Shutdown ein.

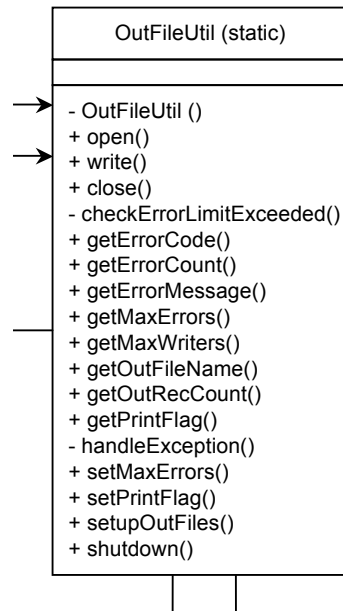
### **handleException()**

dient dazu, die fehlerbeschreibenden Informationen zu beschaffen, aufzubereiten und auszugeben. Ein Fehlertext kann mitgegeben werden. Falls ein Dateiname zur Verfügung steht, versucht `handleException()`, auch den zugehörigen Zugriffszähler-Wert zu beschaffen und auszugeben.

Vor der Ausgabe des Stack Trace Dump legt `handleException()` eine kurze Pause ein, damit es kein Durcheinander gibt. Andernfalls kommen die Meldungen aus `handleException()` und Trace-Ausgaben vermischelt ins Protokoll, da letztere anscheinend in einem eigenen, nicht synchronisierten Thread erzeugt werden.

Abschliessend überträgt `handleException()` seine Aufgabe an `shutdown()` von `FileRegister` und gibt fehlerbeschreibende Parameter mit.

## 8.8 Klasse OutFileUtil



```
package FileService;
```

```
import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
```

```
/**
 * The static class OutFileUtil wraps all output file operations.
 */
```

```
public final class OutFileUtil
{
    /**
     * dummy constructor for objects of class OutFileUtil
     */
    private OutFileUtil() { }
```

### 8.8.1 open()

```
/**
 * open() checks first if the error limit is exceeded (==> shut down);
 * otherwise it verifies the fileID plus the fileName superficially. Then it
 * asks the FileRegister whether a BufferedWriter entry exists for the fileID.
 * If the file is not already registered, a file output stream is opened,
 * using the output file name. Otherwise the request is rejected.
 * After successfully creating the output stream, an output stream writer
```

```

* is created with or without a certain encoding:
* - A null encoding string reference means: no encoding.
* - A non-null encoding string reference means: use it in the call.
* Finally, a buffered writer object is created and its reference plus
* other attributes are stored in the associated FileRegister entry.
*/
public static int open(int    fileID,
                      String fileName,
                      String encoding)
{
    FileOutputStream  outputStream = null;
    OutputStreamWriter streamWriter = null;
    String            errorMsg;

    checkErrorLimitExceeded();

    if (fileID < 0 || fileID >= FileRegister.getMaxWriters())
    {
        errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
            " (" + (FileRegister.getMaxWriters()-1) +
            "): >" + fileID + "< (Output OPEN)";

        return FileRegister.handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
                                          errorMsg);
    }

    if (fileName == null)
    {
        errorMsg = FS_Code.MISSING_FILE_NAME.getMessage() +
            " for ID >" + fileID + "< (Output OPEN)";

        return FileRegister.handleResults(FS_Code.MISSING_FILE_NAME.getCode(),
                                          errorMsg);
    }

    BufferedWriter writer = FileRegister.getWriterRef(fileID);

    if (writer != null)
    {
        errorMsg = FS_Code.OPENED_TWICE.getMessage() +
            " for ID >" + fileID + "< (Output OPEN)";

        return FileRegister.handleResults(FS_Code.OPENED_TWICE.getCode(),
                                          errorMsg);
    }

    try {

```

```

        outputStream = new FileOutputStream(fileName);
    }
    catch(Exception ex) {
        handleException(fileID, null, 'O', ex,
            FS_Code.OUTPUT_OPEN_FAILED.getCode(),
            FS_Code.OUTPUT_OPEN_FAILED.getMessage());
    }

    if (encoding == null)
    {
        try {
            streamWriter = new OutputStreamWriter(outputStream);
        }
        catch(Exception ex) {
            handleException(fileID, fileName, 'O', ex,
                FS_Code.OUTPUT_SW_NOT_CREATED.getCode(),
                FS_Code.OUTPUT_SW_NOT_CREATED.getMessage());
        }
    }
    else
    {
        try {
            streamWriter = new OutputStreamWriter(outputStream, encoding);
        }
        catch(Exception ex) {
            handleException(fileID, fileName, 'O', ex,
                FS_Code.OUT_ENCOD_UNSUPPORTED.getCode(),
                FS_Code.OUT_ENCOD_UNSUPPORTED.getMessage());
        }
    }
}

writer = new BufferedWriter(streamWriter);

if (FileRegister.register(fileID, writer, fileName) != FS_Code.OK.getCode())
{
    shutdown(FS_Code.INTERNAL_FATAL_ERROR.getCode(),
        FS_Code.INTERNAL_FATAL_ERROR.getMessage());
}

return FileRegister.handleResults(FS_Code.OK.getCode(),
    FS_Code.OK.getMessage());
}

```

## 8.8.2 write()

```

/**
 * write() writes an output record from its String parameter.
 */
public static int write(int fileID, String outputString)
{
    String errorMsg;

    // System.out.println("\t outputString = " + outputString);

    checkErrorLimitExceeded();

    if (fileID < 0 || fileID >= FileRegister.getMaxWriters())
    {
        errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
            " (" + (FileRegister.getMaxWriters()-1) +
            "): >" + fileID + "< (WRITE)";

        return FileRegister.handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
            errorMsg);
    }

    BufferedWriter writer = FileRegister.getWriterRef(fileID);

    if (writer == null)
    {
        errorMsg = FS_Code.FILE_NOT_OPEN.getMessage() +
            " with ID >" + fileID + "< (WRITE)";

        return FileRegister.handleResults(FS_Code.FILE_NOT_OPEN.getCode(),
            errorMsg);
    }

    try {
        writer.write(outputString);
    }
    catch(Exception ex) {
        handleException(fileID, null, 'W', ex,
            FS_Code.OUTPUT_WRITE_FAILED.getCode(),
            FS_Code.OUTPUT_WRITE_FAILED.getMessage());
    }

    try {
        writer.newLine();
    }
    catch(Exception ex) {

```

```

        handleException(fileID, null, 'W', ex,
                        FS_Code.OUTPUT_NEWLINE_FAILED.getCode(),
                        FS_Code.OUTPUT_NEWLINE_FAILED.getMessage());
    }

    if (FileRegister.incOutRecCount(fileID) != FS_Code.OK.getCode())
    {
        shutdown(FS_Code.INTERNAL_FATAL_ERROR.getCode(),
                FS_Code.INTERNAL_FATAL_ERROR.getMessage());
    }

    return FileRegister.handleResults(FS_Code.OK.getCode(),
                                    FS_Code.OK.getMessage());
}

```

### 8.8.3 close()

```

/**
 * close() closes the output file.
 */
public static int close(int fileID)
{
    String errorMsg;

    checkErrorLimitExceeded();

    if (fileID < 0 || fileID >= FileRegister.getMaxWriters())
    {
        errorMsg = FS_Code.ILLEGAL_FILE_ID.getMessage() +
            " (" + (FileRegister.getMaxWriters()-1) +
            "): >" + fileID + "< (Output CLOSE)";

        return FileRegister.handleResults(FS_Code.ILLEGAL_FILE_ID.getCode(),
                                        errorMsg);
    }

    BufferedWriter writer = FileRegister.getWriterRef(fileID);

    if (writer == null)
    {
        errorMsg = FS_Code.FILE_NOT_OPEN.getMessage() +
            " with ID >" + fileID + "< (Output CLOSE)";

        return FileRegister.handleResults(FS_Code.FILE_NOT_OPEN.getCode(),
                                        errorMsg);
    }
}

```

```

    try {
        writer.close();
    }
    catch(Exception ex) {
        handleException(fileID, null, 'C', ex,
            FS_Code.OUTPUT_CLOSE_FAILED.getCode(),
            FS_Code.OUTPUT_CLOSE_FAILED.getMessage());
    }

    if (FileRegister.register(fileID, writer) != FS_Code.OK.getCode())
    {
        shutdown(FS_Code.INTERNAL_FATAL_ERROR.getCode(),
            FS_Code.INTERNAL_FATAL_ERROR.getMessage());
    }

    return FileRegister.handleResults(FS_Code.OK.getCode(),
        FS_Code.OK.getMessage());
}

```

### 8.8.4 checkErrorLimitExceeded()

```

/**
 * checkErrorLimitExceeded() obtains the error limit & count values.
 * If the error limit is non-negative and the error count exceeds it,
 * the shutdown sequence is activated.
 */
private static void checkErrorLimitExceeded()
{
    int maxErrors = FileRegister.getMaxErrors();

    if (maxErrors >= 0 && FileRegister.getErrorCount() > maxErrors)
    {
        shutdown(FS_Code.ERROR_LIMIT_EXCEEDED.getCode(),
            FS_Code.ERROR_LIMIT_EXCEEDED.getMessage() +
            ": >" + maxErrors + "<");
    }
}

```

### 8.8.5 getXXX() methods

```

/**
 * The getXXX() methods encapsulate the corresponding FileRegister methods.
 */
public static int getErrorCode()
{ return FileRegister.getErrorCode(); }

public static int getErrorCount()

```

```

{ return FileRegister.getErrorCount(); }

public static String  getErrorMessage()
{ return FileRegister.getErrorMessage(); }

public static int     getMaxErrors()
{return FileRegister.getMaxErrors(); }

public static int     getMaxWriters()
{ return FileRegister.getMaxWriters(); }

public static String  getOutFileName(int fileID)
{ return FileRegister.getOutFileName(fileID); }

public static long    getOutRecCount(int fileID)
{ return FileRegister.getOutRecCount(fileID); }

public static boolean getPrintFlag()
{ return FileRegister.getPrintFlag(); }

```

### 8.8.6 handleException()

```

/**
 * handleException() combines the terminating actions.
 */
private static void handleException(int      fileID,
                                     String    fileName,
                                     char      accessMode,
                                     Exception  excp,
                                     int       exitCode,
                                     String    errorMsg)
{
    String outFileName;

    System.out.println(errorMsg);

    if (fileName == null)
        { outFileName = FileRegister.getOutFileName(fileID); }
    else
        { outFileName = fileName; }

    if (outFileName != null)
    {
        System.out.println("Output file name = " + outFileName);

        long crtCount = FileRegister.getOutRecCount(fileID);
    }
}

```

```

        if (crtCount >= 0)
            { System.out.println("Records written = " + crtCount); }
    }

    // pause the thread: let the print output complete
    try {
        Thread.sleep(100);
    }
    catch (Exception ex) { }

    excp.printStackTrace();

    FileRegister.shutdown(fileID, 'O', accessMode, exitCode, null);
}

```

### 8.8.7 setXXX() methods

8

```

/**
 * The setXXX() and shutdown() methods encapsulate the corresponding
 * FileRegister methods.
 */
public static void setMaxErrors(int errorLimit)
{ FileRegister.setMaxErrors(errorLimit); }

public static void setPrintFlag(boolean errMsgPrint)
{ FileRegister.setPrintFlag(errMsgPrint); }

public static int setupOutFiles(int writeFiles)
{
    checkErrorLimitExceeded();

    return FileRegister.initWriters(writeFiles);
}

```

### 8.8.8 shutdown()

```

public static void shutdown(int    exitCode,
                           String shutdownReason)
{ FileRegister.shutdown(-1, ' ', ' ', exitCode, shutdownReason); }
}

```

#### Anmerkungen

Die Klasse `OutFileUtil` besteht hauptsächlich aus den Methoden `open()`, `write()`, `close()`, `checkErrorLimitExceeded()` und `handleException()`. Alle anderen `OutFileUtil`-Methoden rufen – analog zu `InFileUtil` – lediglich die ihnen korrespondierenden `FileRegister`-Methoden auf. `checkErrorLi-`

`mitExceeded()` und `shutdown()` sind Kopien. Auch `handleException()` weist keinen nennenswerten Unterschied auf. Daher werden nachfolgend nur die drei anfangs genannten Methoden kommentiert.

### **open()**

Eingangs prüft `open()` zuerst, ob das Fehlerlimit überschritten ist. Falls nein, findet eine oberflächliche Prüfung der beiden ersten Parameter statt. Nach bestandener Fehlerlimit- und Parameterprüfung versucht `open()`, sich ein `BufferedWriter`-Zugriffsobjekt vom `FileRegister` zu holen. Falls das gelingt, ist für die gegebene `fileID` bereits eine Datei – normalerweise dieselbe – geöffnet und der Aufruf wird zurückgewiesen.

Andernfalls ist die `fileID` noch – im Array – unbelegt. Zunächst öffnet `open()` einen `FileOutputStream`. Dabei wird unter anderem der Dateiname vom System geprüft und die Verbindung zur Datei hergestellt respektive diese angelegt. Schlägt dies fehl, tritt also eine `Exception` ein, so bricht `open()` den Lauf über `handleException()` kontrolliert ab. Auch alle anderen `Exceptions`, die in `OutFileUtil` auftreten können, enden so.

Nach erfolgreichem Öffnen erzeugt `open()` ein `OutputStreamWriter`-Objekt, das analog zu `InFileUtil` wahlweise ein Encoding oder keines vornimmt.

Der letzte Schritt, um ein für das Schreiben verwendbares Zugriffsobjekt zu beschaffen, besteht im Erzeugen eines `BufferedWriter`-Objekts, dessen Referenz direkt danach zusammen mit dem Dateinamen an die dafür vorgesehene `register()`-Methode übergeben wird. Der Registrierungsschritt muss korrekt ablaufen, weil sonst ein schwerer interner Fehler vorliegt, der zum Shutdown führt.

### **write()**

setzt einen erfolgreichen `open()`-Vorgang voraus. Nach Fehlerlimit- und `fileID`-Prüfung versucht `write()` daher, eine gültige Schreib-Zugriffsobjekt-Referenz von `FileRegister` zu holen. Wenn das fehlschlägt, ist die Datei – noch – nicht geöffnet und der Aufruf wird zurückgewiesen. Andernfalls findet mit der rückgemeldeten Referenz ein Schreibversuch statt, dessen Fehlschlag einen Shutdown auslöst.

Nach erfolgreichem Schreiben folgt ein `newLine()`-Aufruf, damit der aktuelle Ausgabesatz vom nächsten Satz oder vom Dateiende abgegrenzt wird. Nur bei fehlerfreier Ausführung beider Ausgaben wird der Zugriffszähler um 1 erhöht, was unbedingt klappen muss. Abschliessend meldet `write()` den Code 0 – Fehlertext 'OK' – zurück.

### **close()**

Auch `close()` benötigt nach bestandener Fehlerlimit-Prüfung eine gültige `fileID` und eine dadurch bezeichnete Schreib-Zugriffsobjekt-Referenz. Das Verhalten von `close()` ist mit dem Verhalten beim Schließen einer Eingabedatei identisch.