2.4.7 I/O Stream and File Reading

A Java csatornák (stream-ek) segítségével valósítja meg az I/O mveletek nagy részét.

Egy csatorna adatok sorozata, egyik végén befelé a másik végén kifelé "folynak" az adatok.

A Java program szempontjából vannak bemeneti és kimeneti csatornák

A csatornák mellett a Java rendelkezik eszközökkel a közvetlen elérés fájlok és a helyi fájlrendszerek kezeléséhez is.

Bemeneti csatornák

Csak a csatorna elejérl tudunk adatokat elolvasni és levenni

Tartozik hozzá egy adatforrás, ami adatokkal táplálja a csatornát.

Az adatforrás sok minden lehet:

- billentyzet
- fájl
- egy String
- egy program
- egy hálózati végpont
- egy másik csatorna kimenete – stb

Kimeneti csatornák

Csak a csatorna végére tudunk adatokat írni

Tartozik hozzá egy adatnyel, ahova a kiírt adatok mennek

Az adatnyel sok minden lehet:

- képerny
- fájl
- egy String
- egy program
- egy hálózati végpont
- egy másik csatorna bemenete – stb

Csatornák a Java-ban

Ahogyan minden mást, a csatornákat is osztályokkal valósítjuk meg a Java-ban.

Minden csatorna típusnak egy-egy osztály felel meg.

A csatornákat megvalósító osztályok a java.io csomagban találhatók.

Csatornaosztályok csoportosítása

A csatorna iránya alapján vannak:

– bemeneti osztályok

– kimeneti kimeneti

A csatornán folyó adatok típusa szerint vannak:

– bájtcsatorna-osztályok

– karaktercsatorna-osztályok

Feladatuk szerint vannak

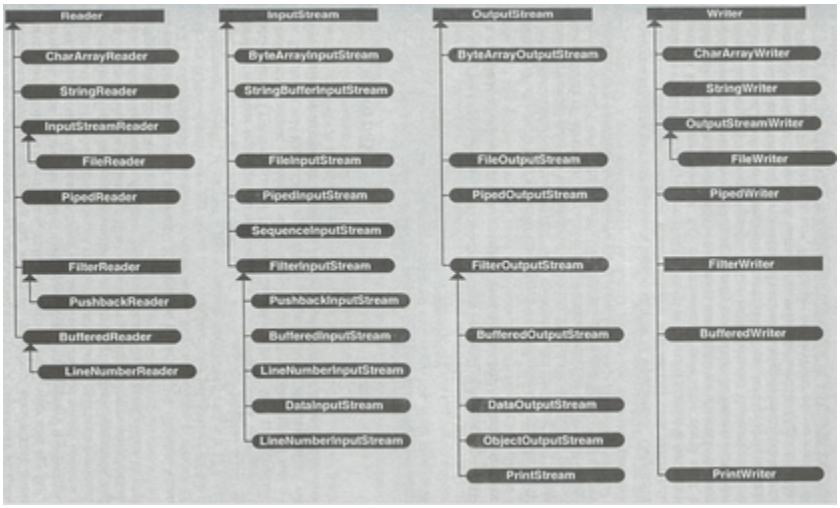
– forrás osztályok

– nyel osztályok

Típus\Irány	Bemeneti	Kimeneti
Bájt	InputStream	OutputStream
Karakter	Reader	Writer

Az ezekbl származó osztályok így végzdnek.

A név végébl tudhatjuk, hogy milyen típusú osztályról van szó. Például: FileInputStream, FileOutputStream, FileReader, FileWriter



Bájt- és karaktercsatornák

A két típus a csatorna által kezelt legkisebb adategység típusában különbözik: – bájt: 8 bit – char: 16 bit, (Unicode)

A két osztálycsalád nagyon sok szempontból hasonlít egymásra.

Amit az egyik típusról megtanultunk, azt a másiknál is használhatjuk értelemszer módosításokkal.

Például: **OutputStream: public void write(byte c[]) throws IOException Writer: public void write(char c[]) throws IOException**

Feladat szerinti csoportosítás

Bemeneti csatornák, forrásosztályok: – Pl. *FileInputStream, FileReader*

Kimeneti csatornák, nyelosztályok: – Pl. *FileOutputStream, FileWriter*

Szrk: – egy meglév csatornát új képességekkel, tulajdonságokkal egészít ki. Például:

– *BufferedInputStream*, *BufferedReader*: az *InputStream*, illetve a *Reader* pufferelt beolvasást biztosító változata.

– A szrket mindig egy létez csatornaobjektum "fölé" hozzuk létre, a konstruktorának átadjuk a csatorna objektumot

Példa a szrkre:

PrintWriter:

– a hozzárendelt csatornát kiegészíti a különböz adattípusok szöveges kiírásának képességével.

– Konstruktorának meg kell adni egy kimeneti csatornát, ezt fogja kiegészíteni ezzel a képességgel.

FileWriter fout = new FileWriter("printwriter.txt");

PrintWriter out = new PrintWriter(fout);

vagy

PrintWriter out = new PrintWriter(new FileWriter("printwriter.txt"));

Csatornák alapfunkciói

Ezeket a funkciókat a négy alaposztály definiálja: – ***InputStream, OutputStream, Reader, Writer***

A többi osztály ezektl örökli ket.

Az alapfunkciók:

– Csatorna megnyitása, lezárása

– Kiíró mveletek

– Olvasó mveletek

– Könyvjelz-mechanizmus

– Egyéb mveletek

Csatornák megnyitása és lezárása

A Java I/O osztályoknak nincsen külön megnyitó metódusa, a konstruktor hozza létre és rögtön meg is nyitja a csatornát.

A csatorna lezárására a close() metódus szolgál.

Kimeneti csatornák esetén a close() végrehajtja a flush() metódust is, azaz lemezre menti a pufferek tartalmát.

Ha lezárt csatornára próbálunk meghívni valamilyen író/olvasó mveletet, akkor IOException kivételt kapunk.

Például:

FileWriter fout = new FileWriter("filenev.txt"); ... // itt lehet használni a csatornát fout.close();

Olvasó mveletek

Mind a bájt, mind a karakter típusú kimeneti csatornák rendelkeznek a következ 3 beolvasó metódussal:

public int read() throws IOException

public int read(byte t[]) throws IOException

public int read(byte t[], int off, int len) throws IOException

karaktercsatornák esetén byte helyett char értend

Az els read metódus 1 vagy 2 bájtot olvas be, majd int-té alakítva adja vissza.

A másik kett annyi adatot olvas, amennyi a megadott tömbben, vagy résztömbben elfér. Visszaadja a beolvasott adategységek (byte, vagy char) számát, vagy -1-et, ha nem volt olvasható adat

Kiíró mveletek

Mind a bájt, mind a karakter típusú kimeneti csatornák rendelkeznek a következ 3 kiíró metódussal:

public void write(int i) throws IOException

public void write(byte t[]) throws IOException

public void write(byte t[], int off, int len) throws IOException

karaktercsatornák esetén byte helyett char értend

Az els write metódus a paraméter 1 vagy 2 legalsó bájtját írja ki.

A karaktercsatornák rendelkeznek a következ 2 metódussal is:

public void write(String s) throws IOException

public void write(String s, int off, int len) throws IOException

Hogyan kezeljük a csatorna végét?

A beolvasó mveletek megkülönböztetik az üres és a véget ért csatornákat.

Üres egy csatorna, ha pillanatnyilag nem tartalmaz adatot. Ilyenkor a beolvasó mveletek addig várakoznak, amíg nem érkezik adat.

Véget ért egy csatorna, ha a csatorna végét jelz jel érkezik meg rajta (fájl vége, UNIX-ban CtrlD, DOS-ban Ctrl-Z billenty)

Melyik a jo valasz?

https://stackoverflow.com/questions/4716503/reading-a-plain-text-file-in-java

In JDK 7, the most important classes for text files are:

- Paths and Path - file locations/names, but not their content.
- Files - operations on file content.
- StandardCharsets and Charset (an older class), for encodings of text files.
- the File.toPath method, which lets older code interact nicely with the newer java.nio API.

In addition, the following classes are also commonly used with text files, for both JDK 7 and earlier versions:

- Scanner - allows reading files in a compact way
- BufferedReader - readLine
- BufferedWriter - write + newLine

When reading and writing text files:

- it's often a good idea to use buffering (default size is 8K)
- there's always a need to pay attention to exceptions (in particular, IOException and FileNotFoundException)

Character Encoding

In order to correctly read and write text files, you need to understand that those read/write operations always use an **implicit character encoding** to translate raw bytes - the 1s and 0s - into text. When a text file is saved, the tool that saves it must always use a character encoding (UTF-8 is recommended). There's a problem, however. The character encoding is not, in general, explicit: it's not saved as part of the file itself. Thus, a program that consumes a text file should know beforehand what its encoding is. If it doesn't, then the best it can do is make an assumption. Problems with encoding usually show up as weird characters in a tool that has read the file.

The FileReader and FileWriter classes are a bit tricky, since they implicitly use the system's default character encoding. If this default is not appropriate, the recommended alternatives are, for example:

```
FileInputStream fis = new FileInputStream("test.txt");
InputStreamReader in = new InputStreamReader(fis, "UTF-8");

FileOutputStream fos = new FileOutputStream("test.txt");
OutputStreamWriter out = new OutputStreamWriter(fos, "UTF-8");

Scanner scanner = new Scanner(file, "UTF-8");
```

Example 1 - JDK 7+

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.List;
```

```
import java.util.Scanner;

public class ReadWriteTextFileJDK7 {

  public static void main(String... aArgs) throws IOException{
    ReadWriteTextFileJDK7 text = new ReadWriteTextFileJDK7();

    //treat as a small file
List<String> lines = text.readSmallTextFile(FILE_NAME);
    log(lines);
    lines.add("This is a line added in code.");
    text.writeSmallTextFile(lines, FILE_NAME);

    //treat as a large file - use some buffering
text.readLargerTextFile(FILE_NAME);
    lines = Arrays.asList("Down to the Waterline", "Water of Love");
    text.writeLargerTextFile(OUTPUT_FILE_NAME, lines);
  }

  final static String FILE_NAME = "C:\\Temp\\input.txt";
  final static String OUTPUT_FILE_NAME = "C:\\Temp\\output.txt";
  final static Charset ENCODING = StandardCharsets.UTF_8;

  //For smaller files

  /**
   Note: the javadoc of Files.readAllLines says it's intended for small
   files. But its implementation uses buffering, so it's likely good
   even for fairly large files.
  */
  List<String> readSmallTextFile(String aFileName) throws IOException {
    Path path = Paths.get(aFileName);
    return Files.readAllLines(path, ENCODING);
  }

  void writeSmallTextFile(List<String> aLines, String aFileName) throws IOException {
    Path path = Paths.get(aFileName);
    Files.write(path, aLines, ENCODING);
  }

  //For larger files

  void readLargerTextFile(String aFileName) throws IOException {
    Path path = Paths.get(aFileName);
    try (Scanner scanner =  new Scanner(path, ENCODING.name())){
      while (scanner.hasNextLine()){
        //process each line in some way
log(scanner.nextLine());
      }
    }
  }

  void readLargerTextFileAlternate(String aFileName) throws IOException {
    Path path = Paths.get(aFileName);
    try (BufferedReader reader = Files.newBufferedReader(path, ENCODING)){
      String line = null;
      while ((line = reader.readLine()) != null) {
        //process each line in some way
log(line);
      }
    }
  }

  void writeLargerTextFile(String aFileName, List<String> aLines) throws IOException {
    Path path = Paths.get(aFileName);
    try (BufferedWriter writer = Files.newBufferedWriter(path, ENCODING)){
      for(String line : aLines){
        writer.write(line);
        writer.newLine();
      }
    }
  }
```

```
  private static void log(Object aMsg){
    System.out.println(String.valueOf(aMsg));
  }

}
```

Example 2 - JDK 7+

This example demonstrates using Scanner to read a file containing lines of structured data. One Scanner is used to read in each line, and a second Scanner is used to parse each line into a simple name-value pair. The Scanner class is only used for reading, not for writing.

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;

/** Assumes UTF-8 encoding. JDK 7+. */
public class ReadWithScanner {

  public static void main(String... aArgs) throws IOException {
    ReadWithScanner parser = new ReadWithScanner("C:\\Temp\\test.txt");
    parser.processLineByLine();
    log("Done.");
  }

  /**
   Constructor.
   @param aFileName full name of an existing, readable file.
  */
  public ReadWithScanner(String aFileName){
    fFilePath = Paths.get(aFileName);
  }


  /** Template method that calls {@link #processLine(String)}.  */
  public final void processLineByLine() throws IOException {
    try (Scanner scanner =  new Scanner(fFilePath, ENCODING.name())){
      while (scanner.hasNextLine()){
        processLine(scanner.nextLine());
      }
    }
  }

  /**
   Overridable method for processing lines in different ways.

   <P>This simple default implementation expects simple name-value pairs, separated by an
   '=' sign. Examples of valid input:
   <tt>height = 167cm</tt>
   <tt>mass =  65kg</tt>
   <tt>disposition =  "grumpy"</tt>
   <tt>this is the name = this is the value</tt>
  */
  protected void processLine(String aLine){
    //use a second Scanner to parse the content of each line
Scanner scanner = new Scanner(aLine);
    scanner.useDelimiter("=");
    if (scanner.hasNext()){
      //assumes the line has a certain structure
String name = scanner.next();
      String value = scanner.next();
      log("Name is : " + quote(name.trim()) + ", and Value is : " + quote(value.trim()));
    }
    else {
      log("Empty or invalid line. Unable to process.");
    }
  }
```

```java
  // PRIVATE
private final Path fFilePath;
  private final static Charset ENCODING = StandardCharsets.UTF_8;

  private static void log(Object aObject){
    System.out.println(String.valueOf(aObject));
  }

  private String quote(String aText){
    String QUOTE = "'";
    return QUOTE + aText + QUOTE;
```

```
  }
}
```

Example run of this class:

```
Name is : 'height', and Value is : '167cm'
Name is : 'mass', and Value is : '65kg'
Name is : 'disposition', and Value is : '"grumpy"'
Name is : 'this is the name', and Value is : 'this is the value'
Done.
```

Example 3 - JDK < 7

The try-with-resources feature is not available prior to JDK 7. In this case, you need to exercise care with respect to the close method:

- it always needs to be called, or else resources will leak
- it will automatically flush the stream, if necessary
- calling close on a "wrapper" stream will automatically call close on its underlying stream
- closing a stream a second time has no consequence
- when called on a Scanner, the close operation only works if the item passed to its constructor implements Closeable. Warning: if you pass a Fileto a Scanner, you will *not* be able to close it! Try using a FileReader instead.

Here's a fairly compact example (for JDK 1.5) of reading and writing a text file, using an explicit encoding. If you remove all references to encoding from this class, it will still work -- the system's default encoding will simply be used instead.

```
import java.io.*;
import java.util.Scanner;

/**
 Read and write a file using an explicit encoding.
 JDK 1.5.
 Removing the encoding from this code will simply cause the
 system's default encoding to be used instead.
*/
public final class ReadWriteTextFileWithEncoding {

  /** Requires two arguments - the file name, and the encoding to use.  */
  public static void main(String... aArgs) throws IOException {
    String fileName = aArgs[0];
    String encoding = aArgs[1];
    ReadWriteTextFileWithEncoding test = new ReadWriteTextFileWithEncoding(
      fileName, encoding
    );
    test.write();
    test.read();
  }

  /** Constructor. */
  ReadWriteTextFileWithEncoding(String aFileName, String aEncoding){
    fEncoding = aEncoding;
    fFileName = aFileName;
  }

  /** Write fixed content to the given file. */
  void write() throws IOException  {
    log("Writing to file named " + fFileName + ". Encoding: " + fEncoding);
    Writer out = new OutputStreamWriter(new FileOutputStream(fFileName), fEncoding);
    try {
      out.write(FIXED_TEXT);
    }
    finally {
      out.close();
    }
  }

  /** Read the contents of the given file. */
  void read() throws IOException {
    log("Reading from file.");
    StringBuilder text = new StringBuilder();
    String NL = System.getProperty("line.separator");
    Scanner scanner = new Scanner(new FileInputStream(fFileName), fEncoding);
    try {
      while (scanner.hasNextLine()){
        text.append(scanner.nextLine() + NL);
      }
    }
    finally{
      scanner.close();
    }
    log("Text read in: " + text);
  }

  // PRIVATE
private final String fFileName;
  private final String fEncoding;
  private final String FIXED_TEXT = "But soft! what code in yonder program breaks?";

  private void log(String aMessage){
    System.out.println(aMessage);
  }
}
```

Example 4 - JDK < 7

This example uses FileReader and FileWriter, which implicitly use the system's default encoding. It also uses buffering. To make this example compatible with JDK 1.4, just change StringBuilder to StringBuffer:

```
import java.io.*;

/** JDK 6 or before. */
public class ReadWriteTextFile {

  /**
  * Fetch the entire contents of a text file, and return it in a String.
  * This style of implementation does not throw Exceptions to the caller.
  *
  * @param aFile is a file which already exists and can be read.
  */
  static public String getContents(File aFile) {
    //...checks on aFile are elided
StringBuilder contents = new StringBuilder();

    try {
      //use buffering, reading one line at a time
//FileReader always assumes default encoding is OK!
BufferedReader input =  new BufferedReader(new FileReader(aFile));
      try {
        String line = null; //not declared within while loop
/*
        * readLine is a bit quirky :
        * it returns the content of a line MINUS the newline.
        * it returns null only for the END of the stream.
        * it returns an empty String if two newlines appear in a row.
        */
        while (( line = input.readLine()) != null){
          contents.append(line);
          contents.append(System.getProperty("line.separator"));
        }
      }
      finally {
        input.close();
      }
    }
    catch (IOException ex){
      ex.printStackTrace();
    }

    return contents.toString();
  }

  /**
  * Change the contents of text file in its entirety, overwriting any
  * existing text.
  *
  * This style of implementation throws all exceptions to the caller.
  *
  * @param aFile is an existing file which can be written to.
  * @throws IllegalArgumentException if param does not comply.
  * @throws FileNotFoundException if the file does not exist.
  * @throws IOException if problem encountered during write.
  */
  static public void setContents(File aFile, String aContents)
                                 throws FileNotFoundException, IOException {
    if (aFile == null) {
      throw new IllegalArgumentException("File should not be null.");
    }
    if (!aFile.exists()) {
      throw new FileNotFoundException ("File does not exist: " + aFile);
    }
    if (!aFile.isFile()) {
      throw new IllegalArgumentException("Should not be a directory: " + aFile);
    }
    if (!aFile.canWrite()) {
      throw new IllegalArgumentException("File cannot be written: " + aFile);
    }
```

```
    //use buffering
Writer output = new BufferedWriter(new FileWriter(aFile));
    try {
      //FileWriter always assumes default encoding is OK!
output.write( aContents );
    }
    finally {
      output.close();
    }
  }

  /** Simple test harness.   */
  public static void main (String... aArguments) throws IOException {
    File testFile = new File("C:\\Temp\\blah.txt");
    System.out.println("Original file contents: " + getContents(testFile));
    setContents(testFile, "The content of this file has been overwritten...");
    System.out.println("New file contents: " + getContents(testFile));
```

```
  }
}
```