Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Programming 2 with Java

Input / Output

Marcel Pfahrer – prm1

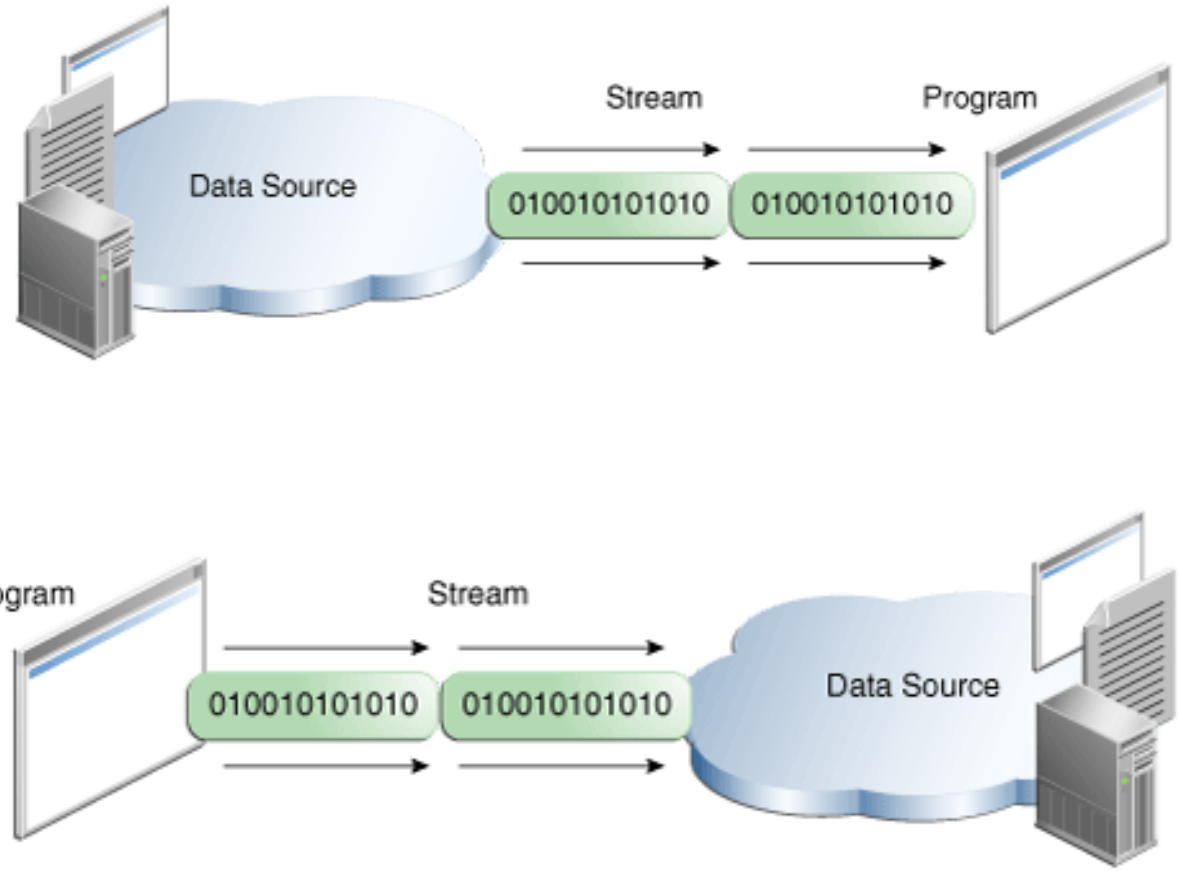
Learning Objectives

Outline – Input / Output

Overview of `java.io.*`

Streams

- A stream is a sequence of data
- An application uses
  - Input streams to read data
  - Output streams to write data



[https://docs.oracle.com/javase/tutorial/essential/io/streams.html]

Byte Streams / Binary Files

- The two classes **`FileInputStream`** and **`FileOutputStream`** are both byte streams
- Reading and writing are done byte by byte
  - `FileInputStream.`**`read`**`(…)` 3 overloads
  - FileOutputStream.**write**(…) 3 overloads
- The application is responsible
  - to get a byte representation of the data to write,
  - to recognize the correct object / data format to be read, and
  - to transform the bytes read from the stream back to a useful object

Character Streams / Text Streams

- A text stream simply is a byte stream where characters are encoded in bytes
  - Reading and writing work correctly only when encoding and decoding match, i.e. use the same charset
  - Default encoding may differ depending on operating system, configuration, and localization settings
- Text reading and writing is done with **`InputStreamReader`** and **`OutputStreamReader`**
  - Both classes simply wrap binary streams and use a (implicit or explicit) charset to encode/decode
  - Both classes have subclasses for reading and writing files

Data Streams

- **`DataInputStream`** and **`DataOutputStream`** support binary I/O of primitive data type values as well as string values
- It is the responsibility of the application to ensure that values are read and written in the correct number and sequence
- The end of an input stream is intercepted by catching the **`EOFException`**
- String values are written and read using **`writeUTF()`** and **`readUTF()`** respectively
  - Encoding is done in a modified form of UTF-8

Object Streams

- Object streams (**`ObjectInputStream`** and **`ObjectOutputStream`**) support I/O of objects
  - An object stream is a serialization of the values contained in the objects
  - The serialized objects are written to, resp. read from data streams
  - Objects are serializable / deserializable if they implement the marker interface **`Serializable`**
- Correct writing and reading is only guaranteed if both operations use identical versions of all classes

- Note: Due to the inherent problems with portability and schema evolution, using object streams is not recommended as a good practice for object persistence

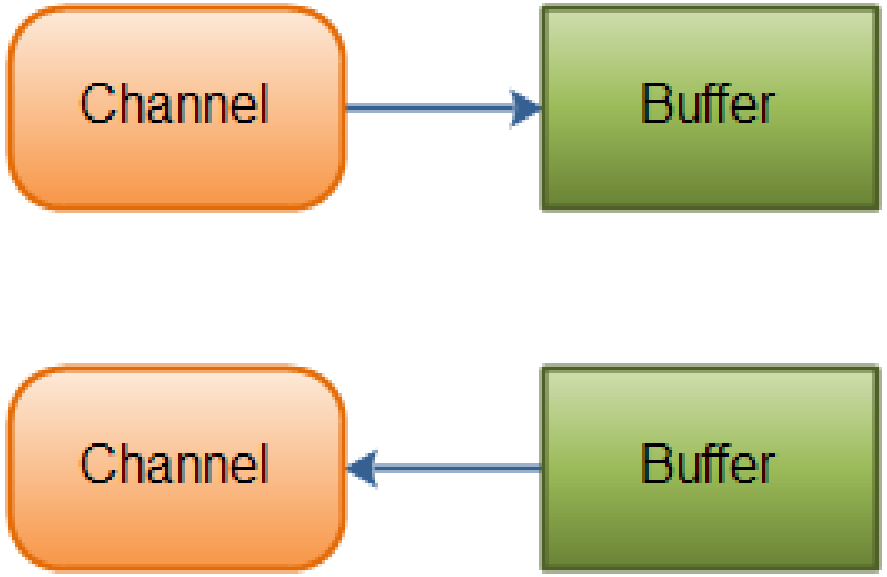
Introduction to `java.nio.*`

Motivation and Overview

- Java NIO (New IO) was introduced with Java 1.4 as an alternative API to standard Java IO and Java networking
- Main differences are
  - Buffer oriented IO (instead of stream oriented)
  - Non blocking IO
  - Selectors
- Java NIO is the better choice for modern applications with different data sources
  - Responsive user interfaces
  - Use of multicore systems
- The central abstractions of the NIO APIs are:
  - Buffers
  - Charsets
  - Channels
  - Selectors

Channels and Buffers

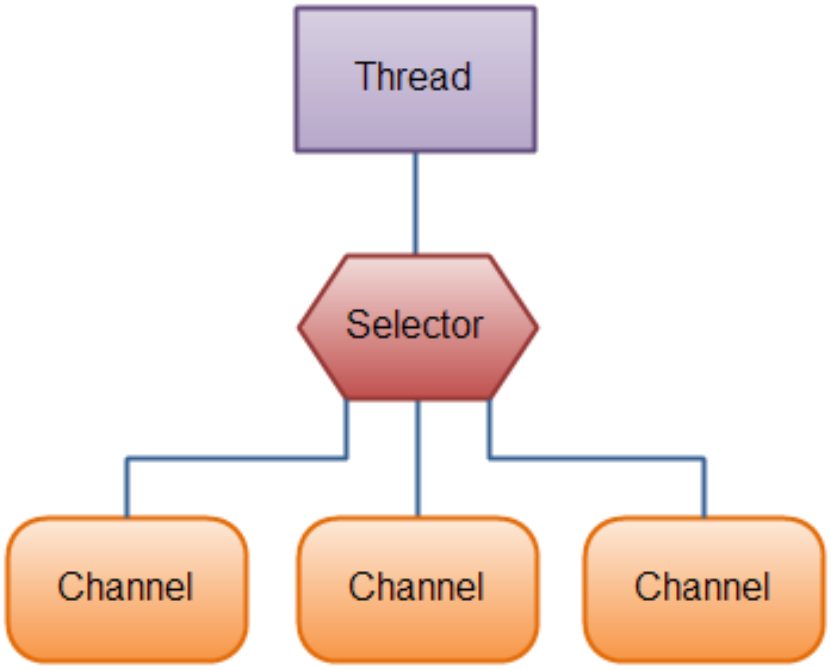
- Typically, all IO in NIO starts with a channel
- A channel is a bit like a stream, but data is read into and written from buffers
- The primary channel implementations are
  - `FileChannel`
  - `DatagramChannel`
  - `SocketChannel`
  - `ServerSocketChannel`
- Buffers cover the basic data type to be sent via IO:
  - `ByteBuffer`
  - `CharBuffer`
  - `DoubleBuffer`
  - `FloatBuffer`
  - `IntBuffer`
  - ...



[http://tutorials.jenkov.com/java-nio/overview.html]

Selectors

- A selector allows a single thread to handle multiple channels
- This is mainly useful when working with network I/O



[http://tutorials.jenkov.com/java-nio/overview.html]

`java.nio` package

- Java NIO is provided in package **`java.nio`** and related subpackes
- Important packages are
  - **`java.nio`** base package and contains buffers
  - **`java.nio.channels`** channels, selectors, and pipes
  - **`java.nio.charset`** charset and related encoding and decoding classes
  - **`java.nio.file`** interfaces and classes to access file systems
  - **`java.nio.file.attribute`** interfaces and classes for file and file system attributes

Working with Files and Folders

`Path` objects

- A path represents the location of a file or directory in a file system
- An instance of the class **`java.nio.file.Path`** represents a file system path
  - Instances are created using the **`Paths.get()`** methods
  - The path may exist or not
  - It provides information on the referenced file or directory
  - It is used to derive other paths
  - It is used as parameter for many I/O operations
- The class provides methods to be used for testing
  - if the file or directory exists or not
  - whether the referenced object represents a regular file, a directory, or a link file
  - whether the file is readable, writeable, and/or executable
  - …

Navigating in the File System Hierarchy

- Path objects are also used for navigation within a file system:
  - Transform relative to absolute paths and vice versa
  - Navigate to the parent directory
  - Combine paths to navigate to child directory or contained files
  - Derive sub-paths
  - …

`Files` objects

- The **`java.nio.file.Files`** class provides all the functionality for manipulating files and directorys:
  - Create, delete, copy, move, rename
  - The static methods take at least one **`Path`** parameter as input

- Other methods of the class are used to open a file for doing I/O
  - Non-bufferd I/O for small files
  - Buffered I/O for larger files

Text File Encoding

Text Files vs. Binary Files

- As already mentioned earlier, text files are binary files where characters are encoded into a series of bytes
- Different encodings have been used in the past for several reasons:
  - Different character sets for different languages
  - Memory size limitations
  - Differences in processor architectures (little endian vs. big endian)
- Current programming languages typically represent a character with 2 bytes, but try to reduce file size by encoding text with less than 2 bytes
- Java NIO provides the **`java.nio.charset.Charset`** class to explicitly specify the character encoding
  - An instance of the class is obtained by calling the **`Charset.forName()`** method
  - This instance is then used for the text I/O methods and objects

Standard Encodings

- A java implementation must provide the following standard encodings:

Charset	Description
`US-ASCII`	Seven-bit ASCII, a.k.a. `ISO646-US`, a.k.a. the Basic Latin block of the Unicode character set
`ISO-8859-1`	ISO Latin Alphabet No. 1, a.k.a. `ISO-LATIN-1`
`UTF-8`	Eight-bit UCS Transformation Format
`UTF-16BE`	Sixteen-bit UCS Transformation Format, big-endian byte order
`UTF-16LE`	Sixteen-bit UCS Transformation Format, little-endian byte order
`UTF-16`	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

Example of Unicode character encoding

- In the table below, UTF-8 and UTF-16 are two different encoding schemata for Unicode characters

Character	UTF-8: hex (dec)	UTF-16: hex (dec)
e	65 (101)	00 65 (0 101)
é	C3 A9 (195 169)	00 E9 (0 233)

Commonly Used Methods for Small Files

Reading small files

- Reading all bytes from a binary file:

```
Path file = Paths.get("./files/image.png");
byte[] image = Files.readAllBytes(file);
System.out.println(image.length + " bytes read");
```

- Reading all lines from a text file:

```
file = Paths.get("./files/goodData.txt");
List<String> lines = Files.readAllLines(file, Charset.forName("UTF-8"));
System.out.println(lines.size() + " lines read");
```

Writing small files

- Writing all bytes to a binary file:

```
Files.write(file, image);
```

- Writing all lines to a text file:

```
Files.write(file, lines, Charset.forName("UTF-8"));
```

Buffered I/O for Text Files

Buffered Reading

- Buffered reading is done using the **`java.io.BufferedReader`** class
- An instance of a **`BufferedReader`** is obtaining by invoking **`Files.newBufferedReader()`**
- The reader is read until a call to **`readLine()`** returns a null reference

```
try (BufferedReader in = Files.newBufferedReader(source, charset)) {

    String line = null;
    BufferedWriter out = null;

    while ((line = in.readLine()) != null) {
        ...
```

Buffered Writing

- A similar class and method is provided for buffered writing:
  - A call to **`Files.newBufferedWriter()`** returns an instance of the **`java.io.BufferedWriter`** class

- Note: When an application terminates, you must ensure that buffered writer objects are closed; otherwise you may loose some data because it is not yet written to the physical storage device

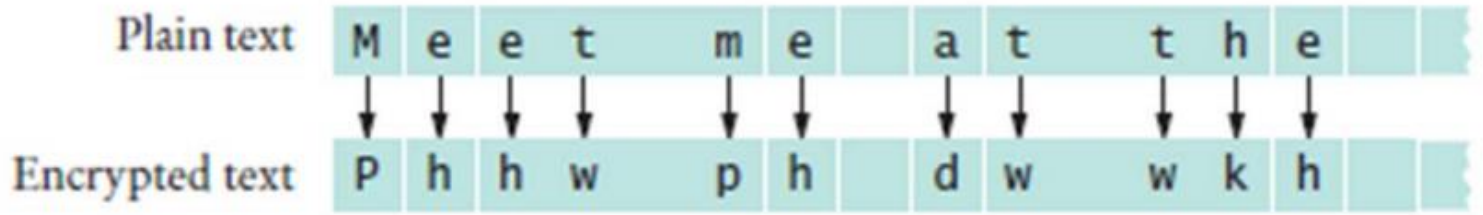
Exercises

File splitting

- Get the file “people.csv”
- It contains comma separated values where
  - each line represents data of a person
  - each line contains 5 values: id, name, firstname, zip, and city
  - lines are ordered by name and firstname
- Read the file and write the data into separate files:
  - Create a separate file for every character A to Z
  - Filenames are the character and the suffix “.split”
  - The files contain all the persons where the name starts with the respective character
  - The files are placed in a directory with the same name as the input file and the suffix “.split”
  - Files and directory may exist or not
    - If they exist, overwrite the existing
    - If not, create them

Caesar Cipher

- Write a simple program that reads text from a text file, encrypts the text using Caesar cipher encryption, and writes the encrypted data to another file
- Implement also functionality to decrypt a previously encrypted file
- Caesar cipher is a simple encryption algorithm:
  - Each character in the text is replaced by another character being *n* positions to the right (or left) in the character table
  - E.g. with n = 3



- Try to apply your program with a binary file, e.g. an image