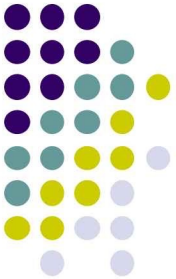
I/O

Ionut

Spalatelu

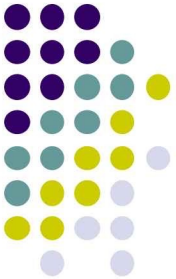
Outline



- I/O streams: byte streams, character streams, buffered streams, scanning and formatting

- File I/O (Featuring NIO.2): Path interface for file operations

Byte streams



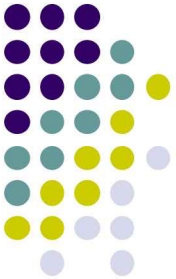
- Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from InputStream and OutputStream.

- There are many byte stream classes but the basic functionality is provided by **FileInputStream** and **FileOutputStream**. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

- They represents a kind of low-level I/O that you should avoid. Since input.txt contains character data, the best approach is to use character streams, as discussed in the next sections. Byte streams should only be used for the most primitive I/O.

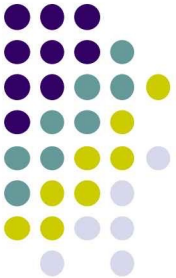
- So why talk about byte streams? Simply because all other stream types are **built on byte streams**.

Character streams



- The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set using the superset of ASCII.
- All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter.
- They use byte streams to perform the physical I/O, while the character stream handles translation between characters and bytes.
- They contain operations to handle Strings directly

Buffered streams



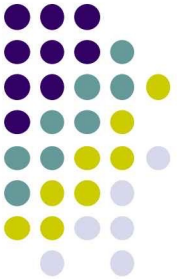
- Handling directly each read or write request by the underlying OS can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

- For reducing this overhead Java has **buffered I/O streams**. Instead of directly communicating with the native API for each byte, buffered input/output streams read/write data from a memory area known as a ***buffer***; the native API being called only when the buffer is empty/full.

- Any character stream can be converted into a buffered stream using the **wrapping** idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class.

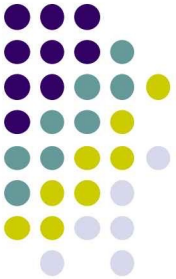
```
reader = new BufferedReader(new FileReader(INPUT_PATH));
writer = new BufferedWriter(new FileWriter(OUTPUT_PATH));
```

Scanning and formatting



- Scanner: provides an useful API for breaking down formatted input into tokens and translating individual tokens according to their data type.
- A tokens can be delimited by using any custom delimiter provided Scanner.useDelimiter(String regex)

Path interface



- The **Path** interface is a programmatic representation of a path in the **file system**. A Path object **contains** the **file name** and **directory list** used to construct the path, and is used to examine, locate, and manipulate files.

- The file or directory corresponding to the Path **might not exist.** You can create a Path instance and manipulate it in various ways: you can append to it, extract pieces of it, compare it to another path.

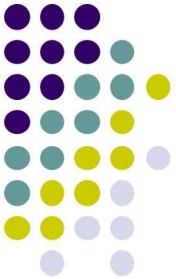
- At the appropriate time, you can use the methods in the **Files** class to check the existence of the file corresponding to the Path, create the file, open it, delete it, change its permissions, and so on.

Path operations using the Path interface

The Path interface includes various methods that can be used to manipulate the target file or directory of the Java object itself in many ways:

- creating a path
- retrieving information about a path
- removing redundancies from a path
- converting a path
- joining two paths
- creating a path between two paths
- comparing two paths

File operations done with Files class



The Files utility class has a bunch of **static** methods for interacting with the target file or directory (of the Path object) that is passed to the method as an argument

- checking a file or directory
- deleting a file or directory
- copying a file or directory
- moving a file or directory
- reading, writing, and creating files
- walking the file tree
- finding files

Managing file metadata

Methods	Comment
size(Path)	Returns the size of the specified file in bytes.
isDirectory(Path, LinkOption)	Returns true if the specified Path locates a file that is a directory.
isRegularFile(Path, LinkOption...)	Returns true if the specified Path locates a file that is a regular file.
isSymbolicLink(Path)	Returns true if the specified Path locates a file that is a symbolic link.
isHidden(Path)	Returns true if the specified Path locates a file that is considered hidden by the file system.
getLastModifiedTime(Path, LinkOption...) setLastModifiedTime(Path, FileTime)	Returns or sets the specified file's last modified time.
getOwner(Path, LinkOption...) setOwner(Path, UserPrincipal)	Returns or sets the owner of the file.
getAttribute(Path, String, LinkOption...) setAttribute(Path, String, LinkOption...)	Returns or sets the value of a file attribute.

Managing commonly used file types

Although the Java libraries provide extensive APIs for dealing with all files in general they don't handle specific types, like Excel files, for example.

- For dealing with CSV files or other Excel types I recommend Apache commons library: **https://commons.apache.org/proper/commons-csv/user-guide.html** **(dependency:** compile group: 'org.apache.commons', name: 'commons-csv', version: '1.8') which handles most of the actions associated with them in an easy way.

Questions