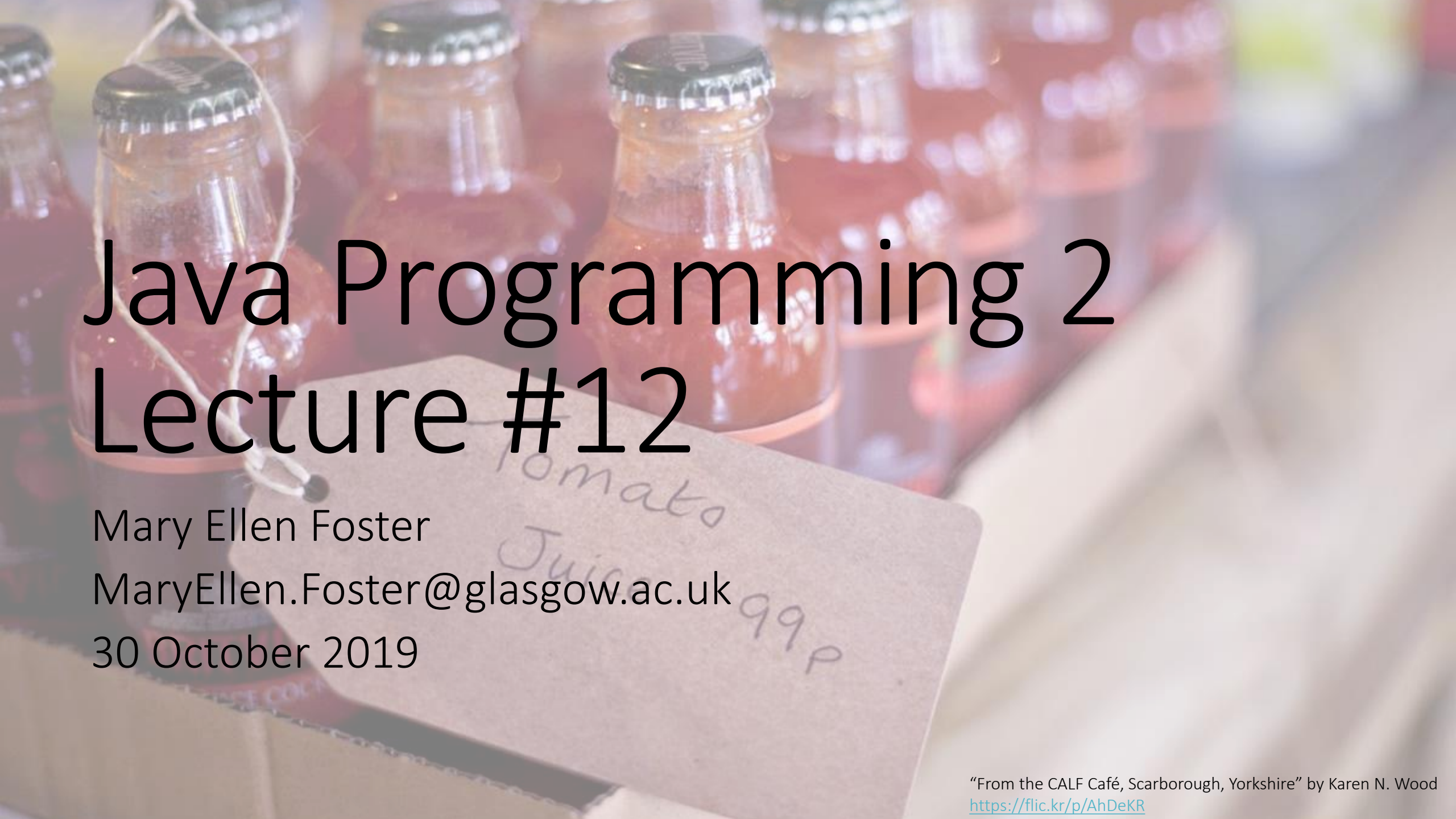



# Java Programming 2
# Lecture #12

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
30 October 2019

"From the CALF Café, Scarborough, Yorkshire" by Karen N. Wood
https://flic.kr/p/AhDeKR

# Outline

Input and output streams

File manipulation with `java.nio`

- `Path`
- `Files`

Style note: returning (Boolean) values

# Packages for input and output

Basic input/output handling classes are in the `java.io` package

Advanced features in the `java.nio` package

"New" since Java 1.4 (revised in Java 1.7)

# Input and output streams

# I/O Streams

Stream: represents an **input source** or an **output destination**

- Disk files
- Devices
- Other programs
- Memory arrays

Data types: bytes, primitive types, characters, objects

Model: A stream is a **sequence of data**

*(Note: nothing to do with Java 1.8 functional Streams (coming soon))*

# Input streams

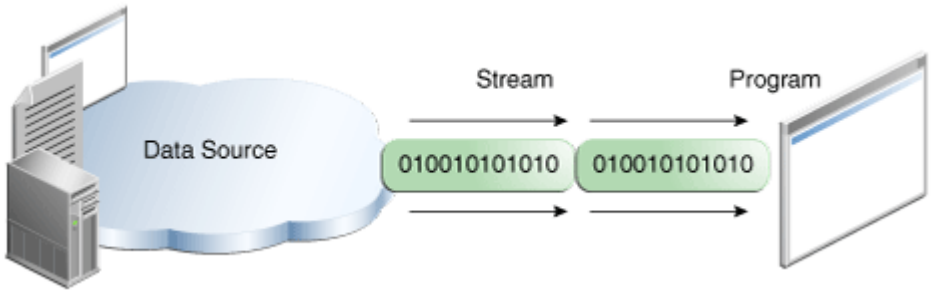
Top-level (abstract) class:
`java.io.InputStream`

Returns a sequence of bytes

Some concrete subclasses:

`FileInputStream`: read bytes from a file on disk

`StringBufferInputStream`: read bytes from a String




*Java tutorial "I/O Streams"*

# Sample program

```
void countBytes(String filename) throws IOException {
    InputStream in = new FileInputStream(filename);
    int total = 0;
    while (in.read() != -1) {
       total++;
    }
    System.out.printf("size of file %s is %d bytes\n", filename, total);
    in.close();
}
```

# Character streams

The basic `InputStream` classes return a sequence of **bytes**

The `Reader` classes return a sequence of **characters**

- `FileReader`: read from a file
- `StringReader`: read from a string
- `InputStreamReader`: convert a byte reader into a character reader
- `BufferedReader`: adds a `readLine()` method

Can be chained:

```
BufferedReader br =
new BufferedReader (new InputStreamReader (System.in));
```

# Sample code

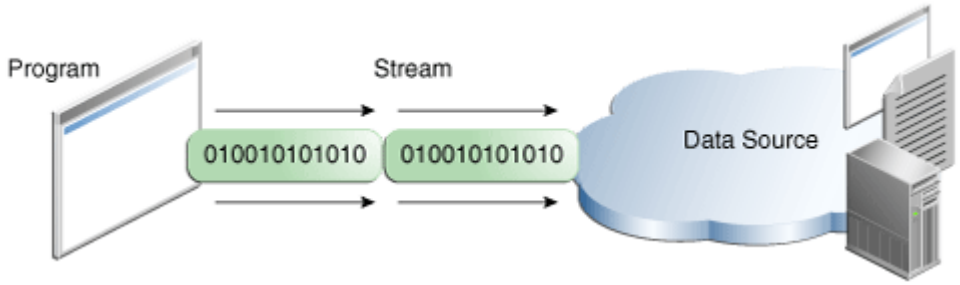
```
void echoFile(String filename) {
        String currentLine = null;
        BufferedReader br = null;
        try {
          br = new BufferedReader(new
FileReader(filename));
          while ((currentLine=br.readLine()) != null)
{
            // echo line to standard output
            System.out.println(currentLine);
          }
        }
        catch (IOException e) {
          e.printStackTrace();
        }
        finally {
          try {
            if (br != null) {
              br.close();
            }
          }
          catch (IOException ee) {
            ee.printStackTrace();
          }
        }
}
```

# Output streams

Byte stream: `OutputStream`

Character stream: `Writer`

- `PrintWriter`: implements `println`, `printf`, `format`
- `FileWriter`: writes to a file
- `StringWriter`: writes to a string
- `OutputStreamWriter`: converts a byte stream to a character stream




*Java tutorial "I/O Streams"*

Can also be chained:

```
PrintWriter pw =
    new PrintWriter (new FileWriter ("log.txt"));
```

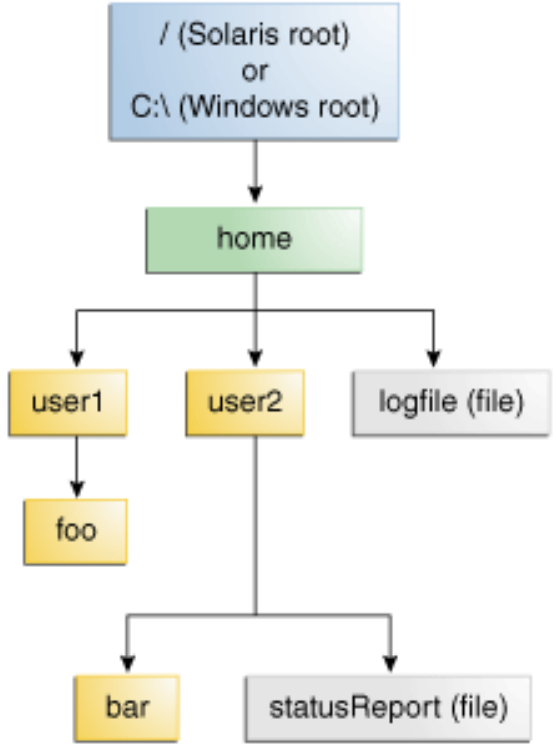
# File I/O with `java.nio` (version 2)

# Basic concept: Path

A ***path*** identifies a file through its location in the file system, beginning from the root node

A ***delimiter*** separates directory names

- Windows: "\"
- Everything else: "/"




*Java tutorial "File I/O"*

# Relative and absolute paths

Absolute path always contains the root element and the complete directory

C:\Users\mefoster\Documents

Relative path needs to be combined with another path to access the file

Lab6\Submission6_1\MyClass.java

# `java.nio.file.Path`

Represents a path in the file system:
- File name
- List of directories to reach it

Used to examine, locate, and manipulate files

Corresponds to the underlying file system – not system independent
- *C:\Users\mefoster\Documents*
- */usr/local/bin/perl*

Corresponding file does not need to exist; you can manipulate a `Path` all you want
- Use methods in `Files` class to deal with actual files

# Operations on `Path`

Syntactic operations: operate on path itself, don't touch file system

Creating a Path:

```
Paths.get ("c:\\users\\joe\\foo");// Paths helper class
```

Retrieving information about a Path:

- Path stores name elements as a sequence
- You can access elements in various ways …

# Access methods

| Method | Return | Notes |
|---|---|---|
| `p.toString()` | `c:\users\joe\foo` | String representation of path |
| `p.getFileName()` | `foo` | Last element in sequence of names |
| `p.getName(0)` | `users` | Path element closest to root |
| `p.getNameCount()` | `3` | Number of elements in path |
| `p.subpath(0,2)` | `users\joe` | Subsequence of path between beginning and ending index |
| `p.getParent()` | `\users\joe` | Path of parent directory |
| `p.getRoot()` | `c:\` | Root of the path |

```
Path p = Paths.get ("c:\\users\\joe\\foo");
```

# Processing Paths

Remove redundancies: use `normalize()`

*c:\users\sally\..\joe\foo -> c:\users\joe\foo*

Does not check the file system – just cleans up the path internally

Converting a Path

`toAbsolutePath()` : prepends current working directory

`toRealPath()` : returns the real path of an existing file

- Converts relative path to absolute path
- Resolves symbolic links (if `true` is passed)
- Removes redundant elements
- Throws an exception if file does not exist (FileNotFoundException) or cannot be accessed (IOException)

# Other notes on Path

It implements `Comparable<Path>`

Collections of Paths can be sorted

It can be used in iteration:

```
Path myPath = …;
for (Path name: myPath) {
    System.out.println (name);
}
```

Also: `equals()`, `beginsWith()`, `endsWith()`

# java.nio.file.Files

Set of static methods for reading, writing, and manipulating files and directories

Methods work on `Path` instances

Most methods throw an `IOException` on I/O failure

# Checking a file or directory

Verify existence:

```
Files.exists (Path), Files.notExists (Path)
```

Check accessibility:

```
Files.isReadable (Path), Files.isWritable (Path),
Files.isExecutable (Path)
```

Do two Paths locate the same file?

```
Files.isSameFile (Path, Path)
```

# Deleting, copying, moving

```
Files.delete (Path)
Files.deleteIfExists (Path)
 // Doesn't throw Exception even if file doesn't exist
Files.copy (Path, Path)
Files.move (Path, Path)
```

# Creating files and directories

`Files.createFile (Path)`

Throws Exception if file already exists, or of parent directory does not exist

`Files.createDirectory (Path)`

Throws Exception if file already exists, or of parent directory does not exist

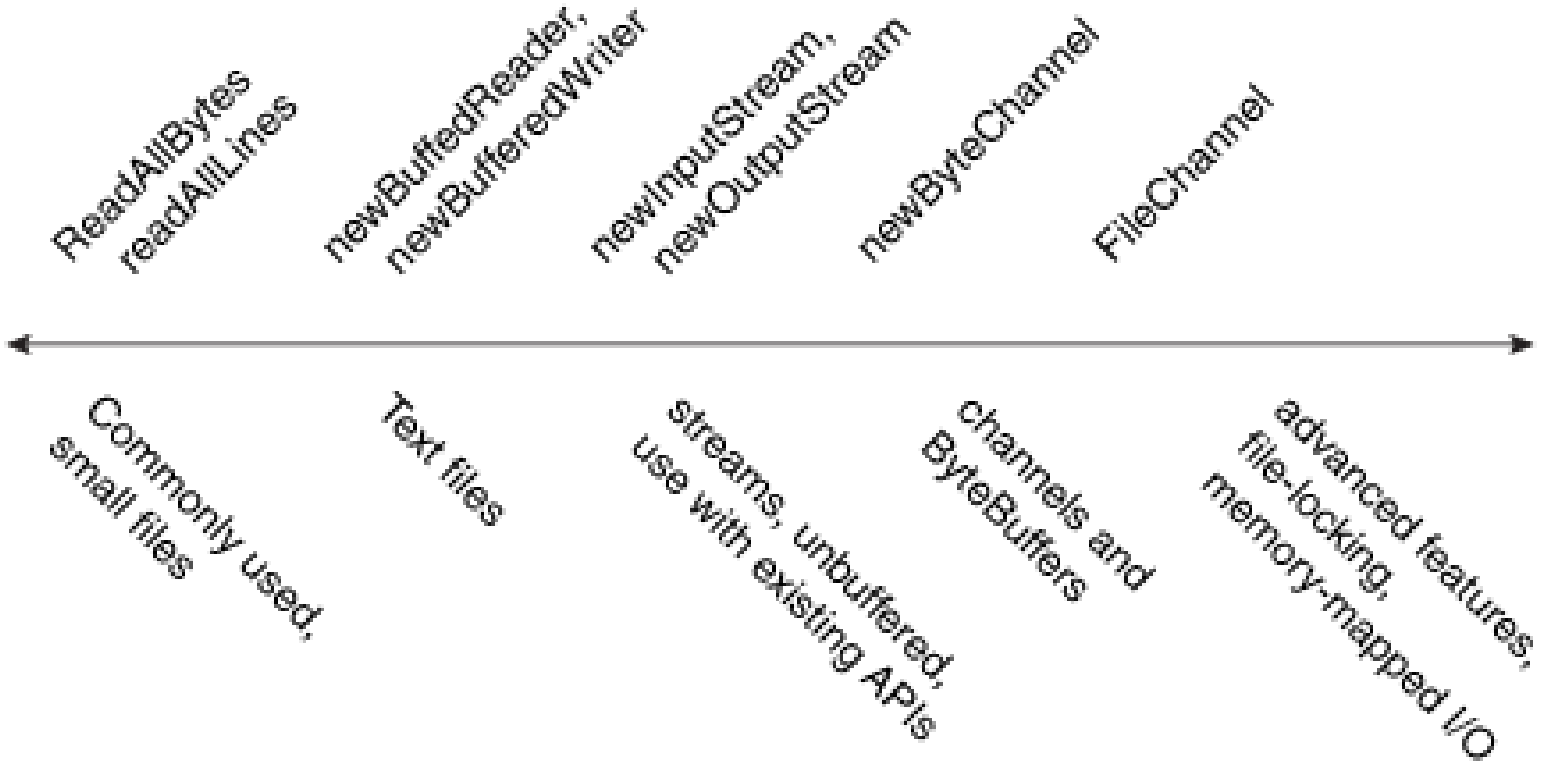
`Files.createDirectories (Path)`

Creates all necessary directories from the top down

# Listing files in a directory

```
Path dir = …;

DirectoryStream<Path> stream =
        Files.newDirectoryStream (dir);

for (Path file : stream) {

    System.out.println (file.getFileName());

}
```

# Reading and writing




*Java tutorial "File I/O"*

# Example code

```
Path p1 = Paths.get ("c:\\Users\\mef\\Documents\\in.txt");
Path p2 = Paths.get ("c:\\Users\\mef\\Documents\\out.txt");
try {
    List<String> lines = Files.readAllLines (p1);
    Files.createFile (p2);
    PrintWriter pw = new PrintWriter (Files.newBufferedWriter (p2));
    for (String line : lines) {
        pw.println (line);
    }
    pw.close();
} catch (IOException ex) {
    ex.printStackTrace();
}
```

# Returning (Boolean) values

# Typical submitted code

```
public boolean isListEmpty() {
    boolean result;
    if (list.isEmpty() == true) {
        result = true;
    } else {
        result = false;
    }
    return result;
}
```

# Modification 1 – just return!

```
public boolean isListEmpty() {
    boolean result;
    if (list.isEmpty() == true) {
        result = return true;
    } else {
        result = return false;
    }
    return result;
}
```

# Modification 2 – remove "== true"

```
public boolean isListEmpty() {
    boolean result;
    if (list.isEmpty() == true) {
        result = return true;
    } else {
        result = return false;
    }
    return result;
}
```

# Modification 3 – remove if/else

```
public boolean isListEmpty() {
    return list.isEmpty();
}
```

# Next time

Overriding equals() and hashCode()

The Objects class

Writing varargs methods