Advanced Programming

Input / Output Streams

File I/O

The Context

- How do we **read from a file**, or **write to a file**:
  - *text:* characters or lines of text, ...?
  - *binary:* arrays of bytes ...?
- How do we **send or receive**:
  - objects in a distributed application?
  - data to/from a serial port?
- How do threads communicate asynchronous?
- How do we write very large sets of data into a relational database?
- ...

I/O Streams

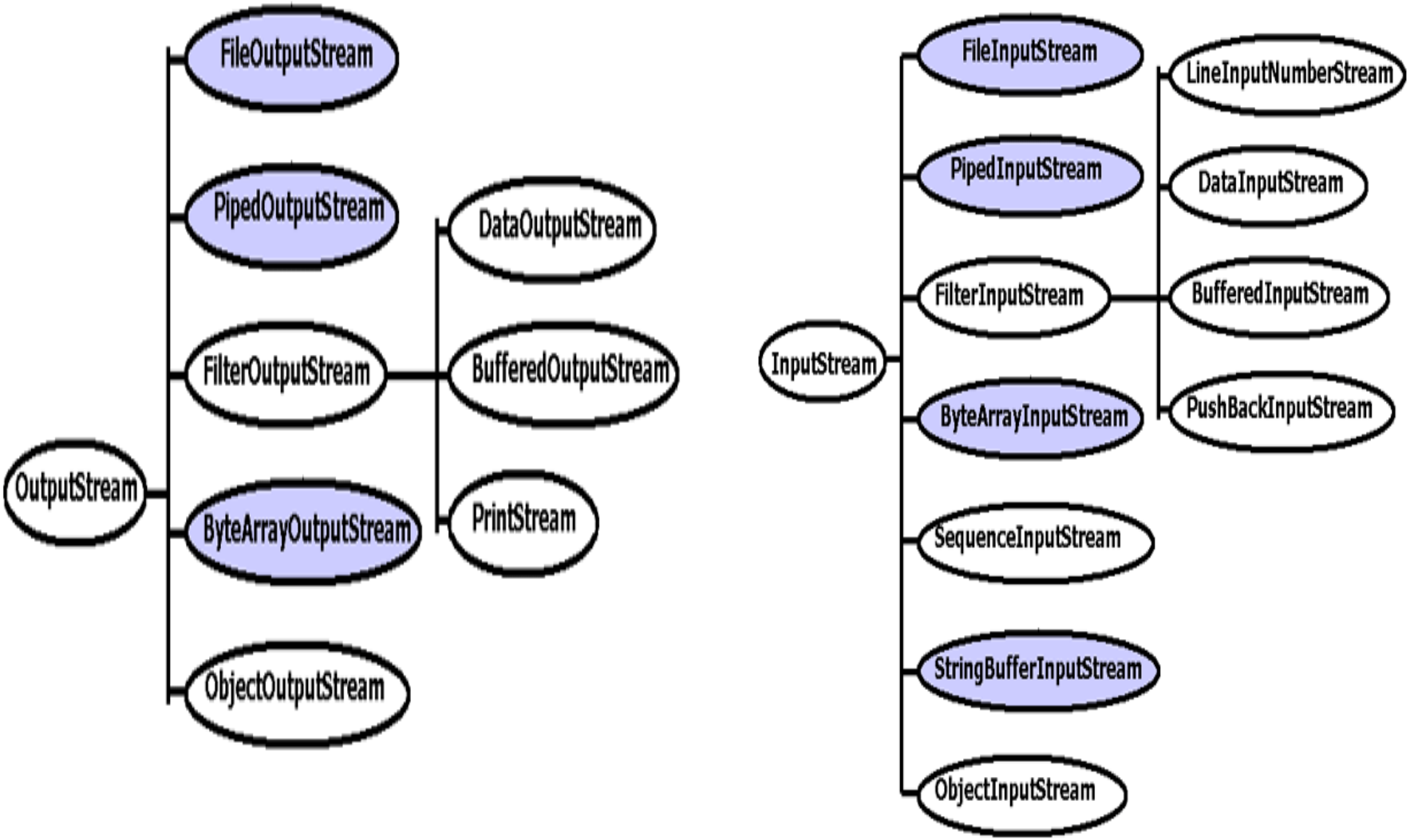
- A stream is a sequence of data: serial, unidirectional, having a source / destination.
- Byte Streams (8 bits) / Char Streams(16 bits)
- *Producers (Output Streams)*

- *Consumers (Input Streams)*

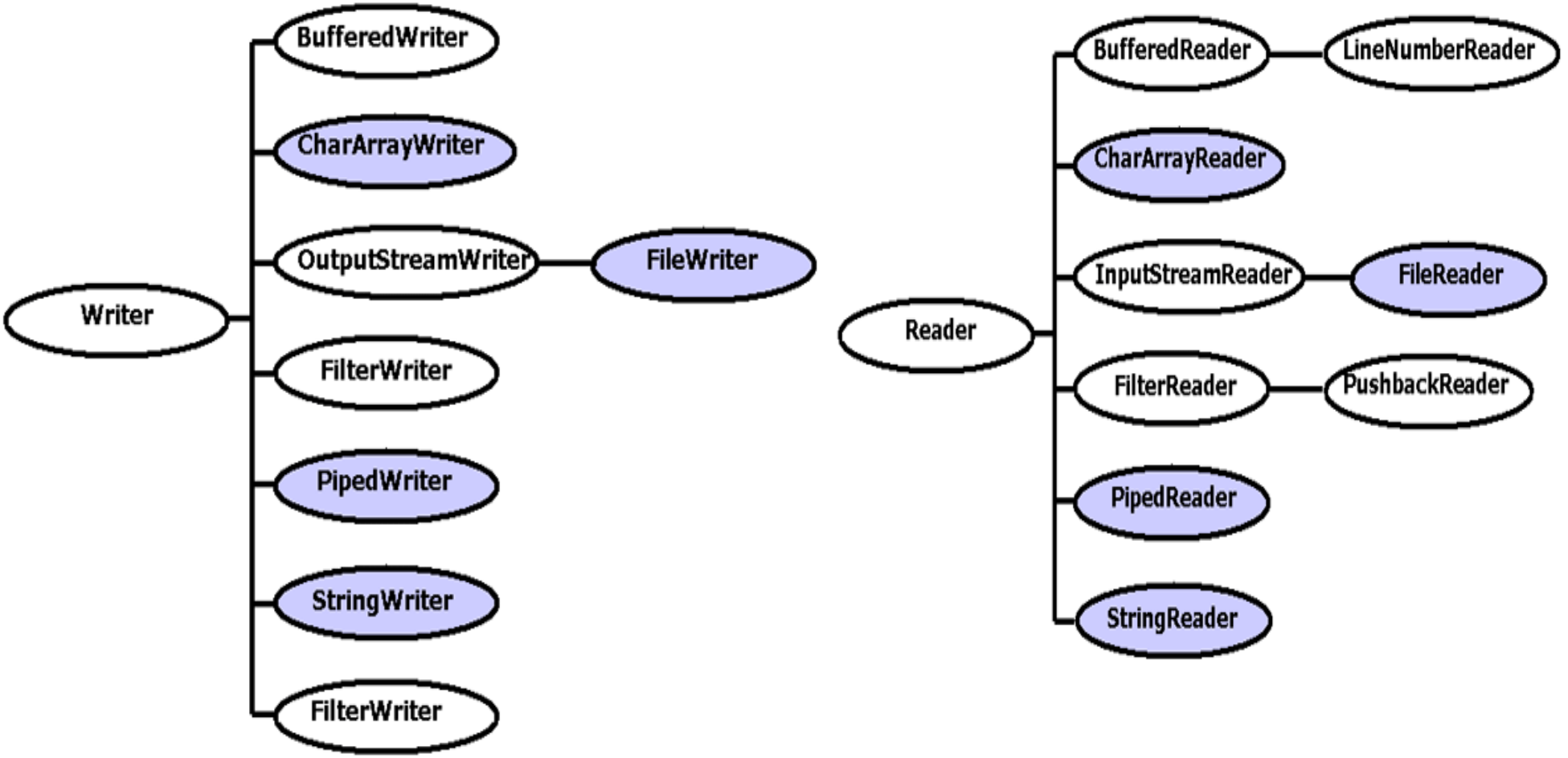
Using I/O Streams

```
import java.io.*;

open the stream; //new StreamClass([arguments]);

while (there is more data to read or write) {

  read/write some data;

  //invoke read(), write() or other specialized methods

}

close the stream; //...dont't forget to invoke close()

take care of the possible exceptions: IOExeption;
```

Byte Streams



Character Streams



Primitive Streams vs. Decorators

- *Primitive* streams "know" to effectively communicate (read or write) with an external "partener" (file, memory, thread, etc.). Examples:
  - `FileReader, FileWriter`
  - `ByteArrayInputStream,ByteArrayOutputStream`
  - `PipedReader, PipedWriter`
- *Decorator* streams "know" to communicate with another stream (primitive or not) in order to process the raw data and offer specialized methods:
  - `BufferedReader, BufferedWriter, PrintWriter`
  - `DataInputStream, DataOutputStream`
  - `ObjectInputStream, ObjectOutputStream`

Creating Streams

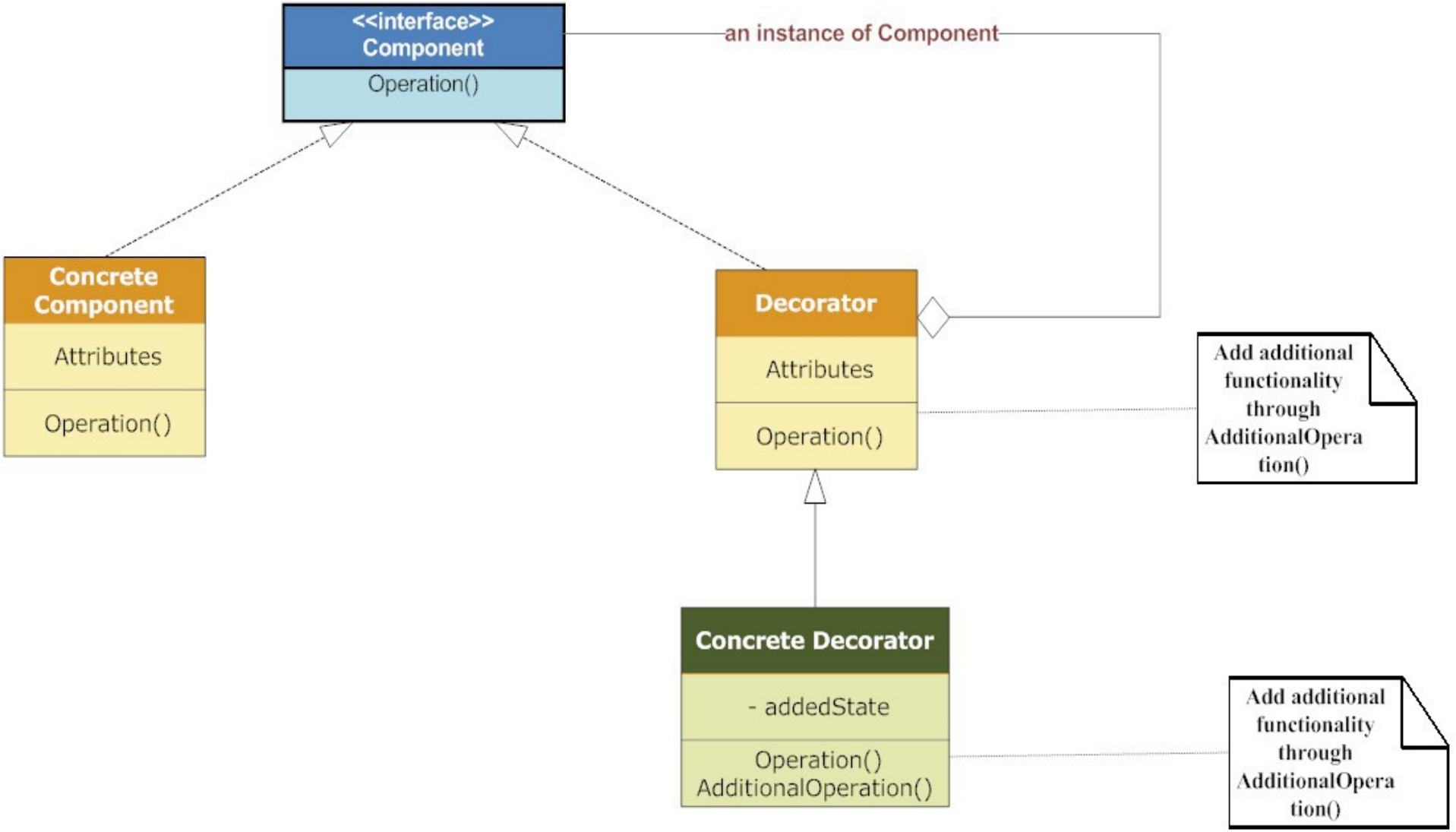
```
PrimitiveStream stream =
        new PrimitiveStream(externalResource);
DecoratorStream decorator =
        new DecoratorStream(stream);
```

Exemplu:

```
FileReader fileReader = new FileReader("fisier.txt");
BufferedReader bufferReader =
    new BufferedReader(fileReader);
String line = bufferedReader.readLine();
```

Decorator Design Pattern

Decorator Pattern Structure



Buffered Streams

BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream

Read /Write data from a stream, buffering elements so as to provide for the efficient reading of arrays, and lines.

```
BufferedOutputStream out = new BufferedOutputStream(
  new FileOutputStream("out.dat"), 1024)
  //1024 is the size of the buffer

for(int i=0; i<100; i++) {
  out.write(i);
  //the buffer is not full yet, the file contains nothing
}

out.flush();
//the buffer is flushed, data is written to the file
```

- Greatly reduces the access to the external resource
- Increases the execution speed

Serializing Primitive Data

DataInputStream, DataOutputStream

- Writing/Reading primtive data in binary format, "*in a machine-independent way*"

```
//Writing
FileOutputStream fos = new FileOutputStream("test.dat");
DataOutputStream out = new DataOutputStream(fos);
out.writeInt(12345);
out.writeDouble(12.345);
out.writeBoolean(true);
out.writeUTF("Sir de caractere");
out.flush();
fos.close();
...
//Reading
FileInputStream fis = new FileInputStream("test.dat");
DataInputStream in  = new DataInputStream(fis);
int i = in.readInt();
double d = in.readDouble();
boolean b = in.readBoolean();
String s = in.readUTF();
fis.close();
```

Serialiazing Objects

ObjectInputStream, ObjectOutputStream

- Writing/Reading objects in a binery format, "*in a machine-independent way*"

```
//Writing
FileOutputStream fos = new FileOutputStream("test.ser");
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject(new Date());
out.writeObject("Hello World");
out.writeInt(12345);
out.flush();
fos.close();

//Reading
FileInputStream fis = new FileInputStream("test.ser");
ObjectInputStream in = new ObjectInputStream(fis);
Date date = (Date)in.readObject();
String message = (String)in.readObject();
int i = in.readInt();
fis.close();
```

Standard I/O Streams

- **`System.in` - `InputStream`**
- **`System.out` - `PrintStream`**
- **`System.err` - `PrintStream`**

Redirecting the standard streams:

`setIn(InputStream)` - redirecting the input

`setOut(PrintStream)` - redirecting the output

`setErr(PrintStream)` - redirecting the error stream

Example:

```
PrintStream fis = new PrintStream(new FileOutputStream("results.txt")));
System.setOut(fis);
PrintStream fis = new PrintStream(new FileOutputStream("errors.txt")));
System.setErr(fis);
```

Scanning and Formatting

java.util.Scanner,

java.util.Formatter, java.text.Format

```
Scanner s = Scanner.create(System.in);
String name = s.next();
int age = s.nextInt();
double salary = s.nextDouble();
s.close();
```

```
System.out.printf("%s %8.2f %n",name, salary, age);
SimpleDateFormat dateFormat =
  new SimpleDateFormat("dd-MM-yyyy");
String date = dateFormat.format(today);
System.out.println("Today in dd-MM-yyyy format : " + date);
```

“Useful” I/O Classes

- *java.io.File*

  An abstract representation of file and directory pathnames.

- *java.io.RandomAccesFile*

  Supports both reading and writing to a file, using a file pointer.

- *java.io.StreamTokenizer*

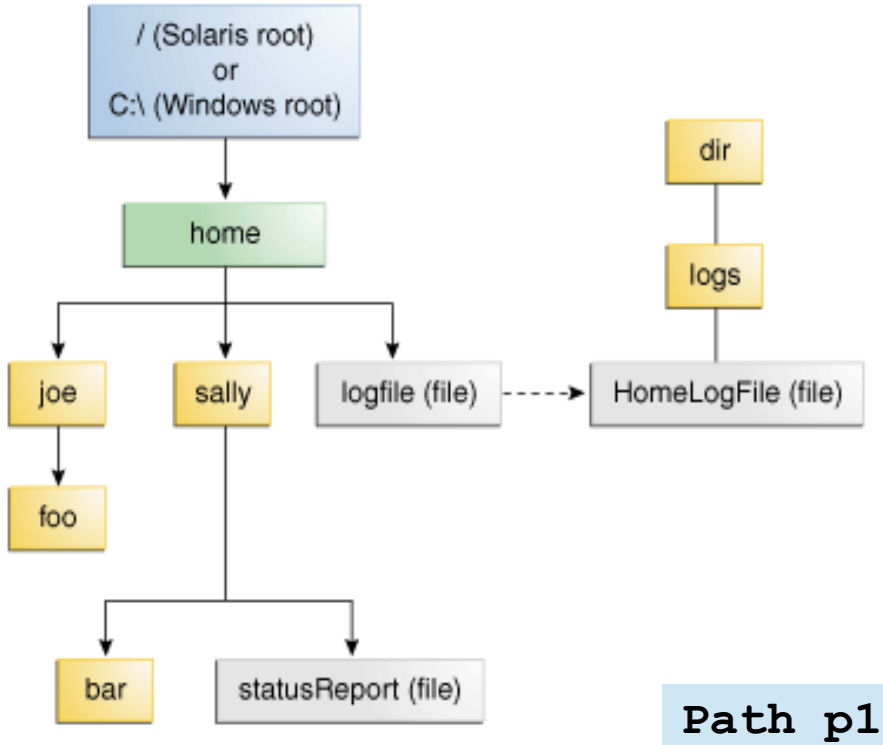
  Takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time.

  (see also *StringTokenizer, String.split*)

- *… here comes the New Java I/O → java.nio*

File I/O (Featuring NIO)

- *java.io.File* has some "issues" (some methods didn't throw exceptions when they failed, no support fo symbolic links, no consistency across platforms, not scalable, no support for recursively walk a file tree, etc.)
- *java.nio.file.Path* is the modern replacement of *File.*



A *Path* object is a programmatic representation of **a path in the file system.** May be:

- relative
- absolute
- symbolic (soft) link
- file
- directory (folder)
- might not exist

```
Path p1 = Paths.get("/home/joe/foo");
```

File Operations

- *java.nio.file.Files* - Static methods that operate on files, directories, or other types of files.
- Examples:

```
boolean readable = Files.isReadable(file);
Files.copy(source, target, REPLACE_EXISTING);
Files.move(source, target, ATOMIC_MOVE);
Files.delete(path);
Path newFile = Files.createFile(path);
Path newDir = Files.createDirectory(path);

BasicFileAttributes attr =
  Files.readAttributes(file, BasicFileAttributes.class);
System.out.println("lastAccessTime: " + attr.lastAccessTime());
...
```

Traversing a Directory Tree

*Files.**walkFileTree**(path, new **FileVisitor**<Path>() { … });*

```
Files.walkFileTree(path, new FileVisitor<Path>() {
  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes
attrs) throws IOException {
    System.out.println("pre visit dir:" + dir);
    return FileVisitResult.CONTINUE;
    // or TERMINATE or SKIP_SIBLINGS or SKIP_SUBTREE
  }
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
throws IOException {
    System.out.println("visit file: " + file);
    return FileVisitResult.CONTINUE;
  }
  public FileVisitResult visitFileFailed(Path file, IOException exc)
throws IOException {
    System.out.println("visit file failed: " + file);
    return FileVisitResult.CONTINUE;
  }
  public FileVisitResult postVisitDirectory(Path dir, IOException exc)
throws IOException {
    System.out.println("post visit directory: " + dir);
    return FileVisitResult.CONTINUE;
  }
});
```

Reading and Writing

- Commonly Used Methods for Small Files

```
byte[] fileArray = Files.readAllBytes(path);
List<String> fileLines = Files.readAllLines(path);
String fileContent = new String(Files.readAllBytes(path));
Files.write(path, buffer); // to write bytes, or lines
```

- Buffered I/O Methods for Text Files

```
BufferedReader reader = Files.newBufferedReader(path);
BufferedWriter writer = Files.newBufferedWriter(path);
```

- Unbuffered Streams

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

- Methods for *Channels* and *ByteBuffers*

While stream I/O reads a character at a time, channel I/O reads a buffer at a time

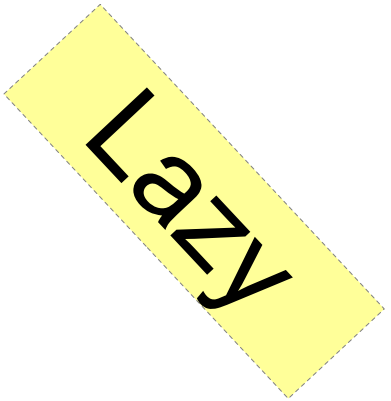
Using Streams and Files

- Iterating over the lines of a text file

```
Path path = Paths.get("c:\\data\\SomeFile.txt");
try (Stream<String> lines = Files.lines(path)) {
   lines.forEachOrdered(line->System.out.println(line));
} catch (IOException e) {
   System.err.println(e);
}
```

- Iterating over the files in a directory

```
Path path = Paths.get("c:\\data");
Files.list(dir)
   .filter((Path file) ->
      file.getFileName().toString().endsWith(".pdf"))
   .forEach(System.out::println);
```



Watching a Directory for Changes

- Watch Service API
- Create and Register

```
WatchService watcher = FileSystems.getDefault().newWatchService();
Path dir = Paths.get("Path/To/Watched/Directory");
dir.register(watcher, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
```

- Watch...

```
while (true) {
    WatchKey key = watcher.take(); // wait for a key to be available
    for (WatchEvent<?> event : key.pollEvents()) {
        WatchEvent.Kind<?> kind = event.kind();
        WatchEvent<Path> ev = (WatchEvent<Path>) event;
        Path fileName = ev.context();

        System.out.println(kind.name() + ": " + fileName);
       key.reset();
    }
}
```