



Object Oriented Architectures and Secure Development

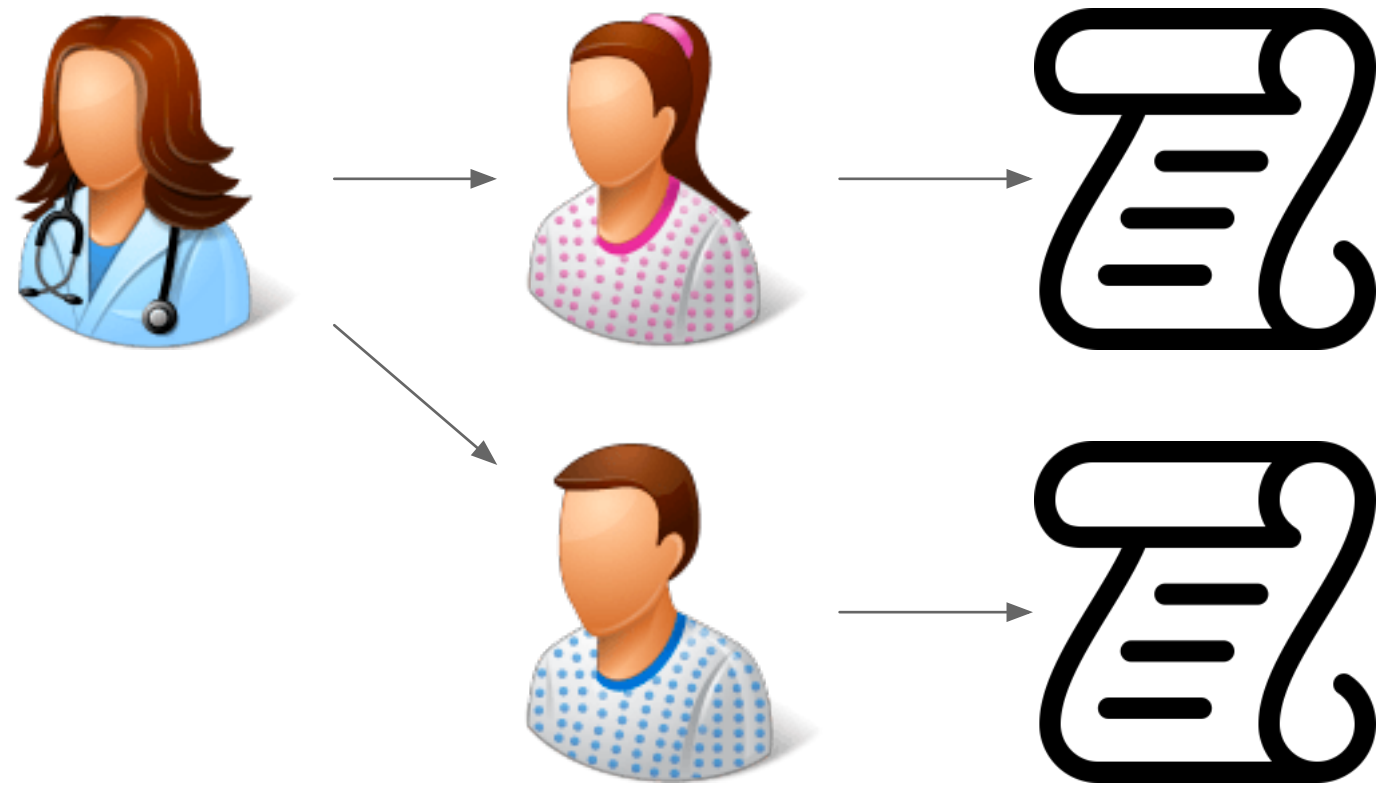
Reading and writing objects from/to file.

Arne Debou

Mattias De Wael

Frédéric Vlummens

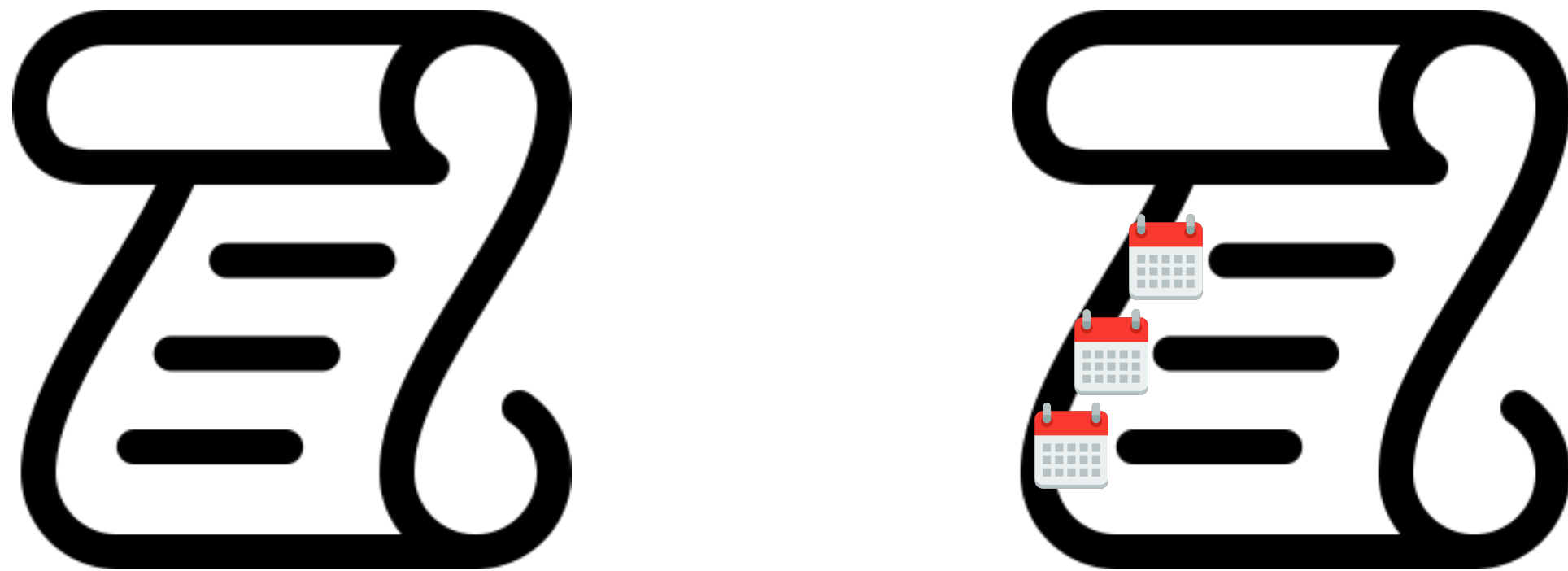
Our hospital system: Where we left of



```
private Map<Patient, String> patients;
```

The diagnosis is just text: String is fine.

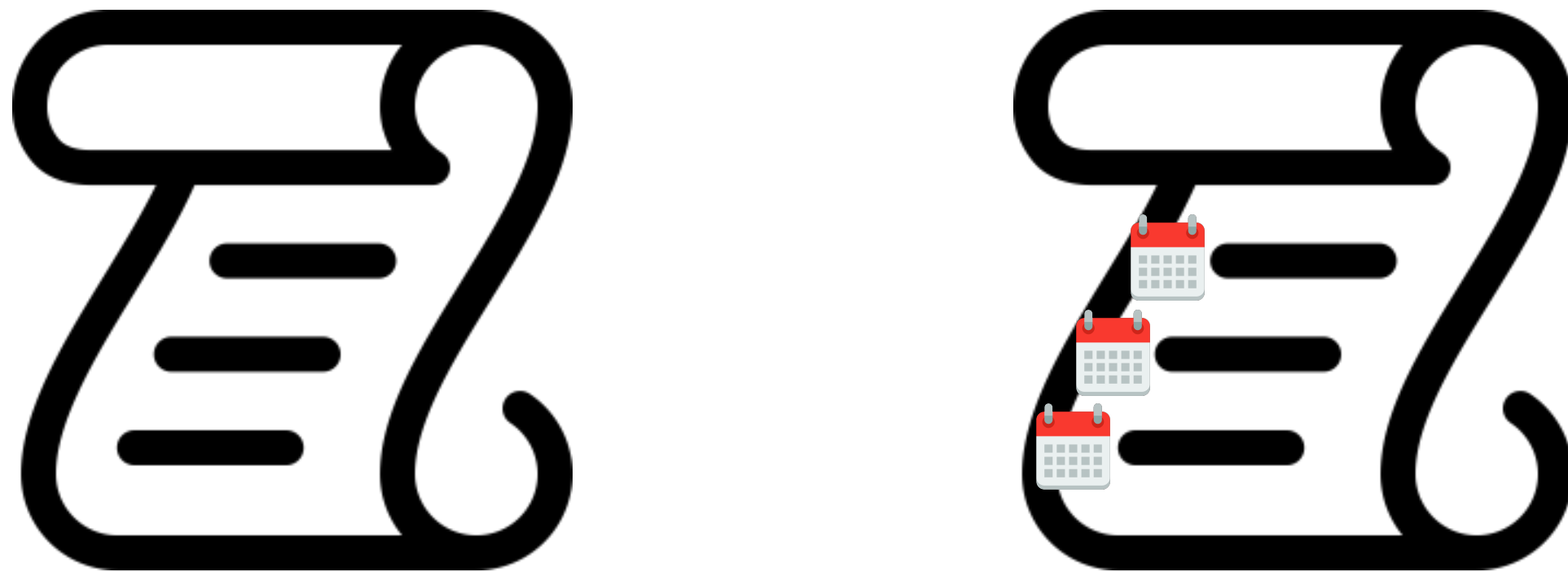
Our hospital system: Where we left of



```
private Map<Patient, Map<Calendar, String> > patients;
```

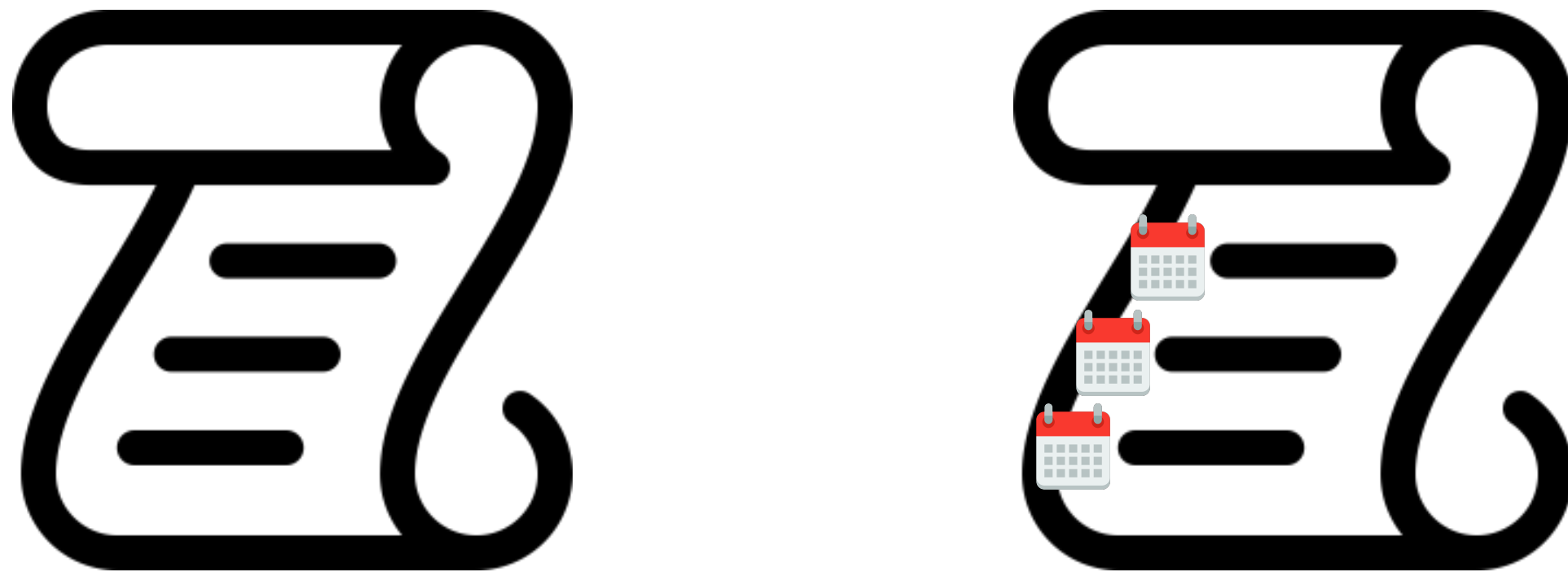
Too complex for a doctor to handle: new class!

Our hospital system: Where we left of



```
private Map<Patient, PatientLog> patients;
```

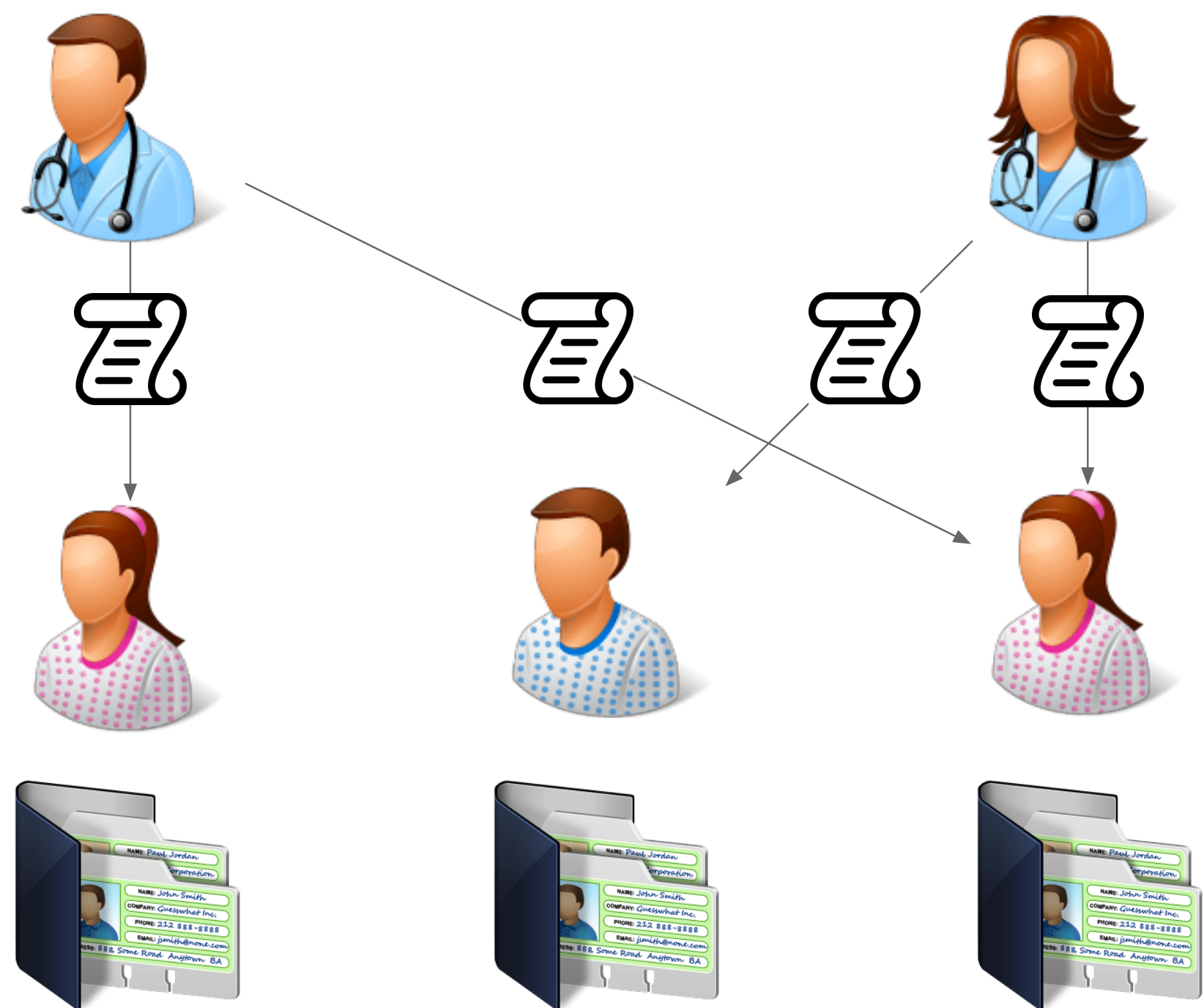
Our hospital system: Where we left of



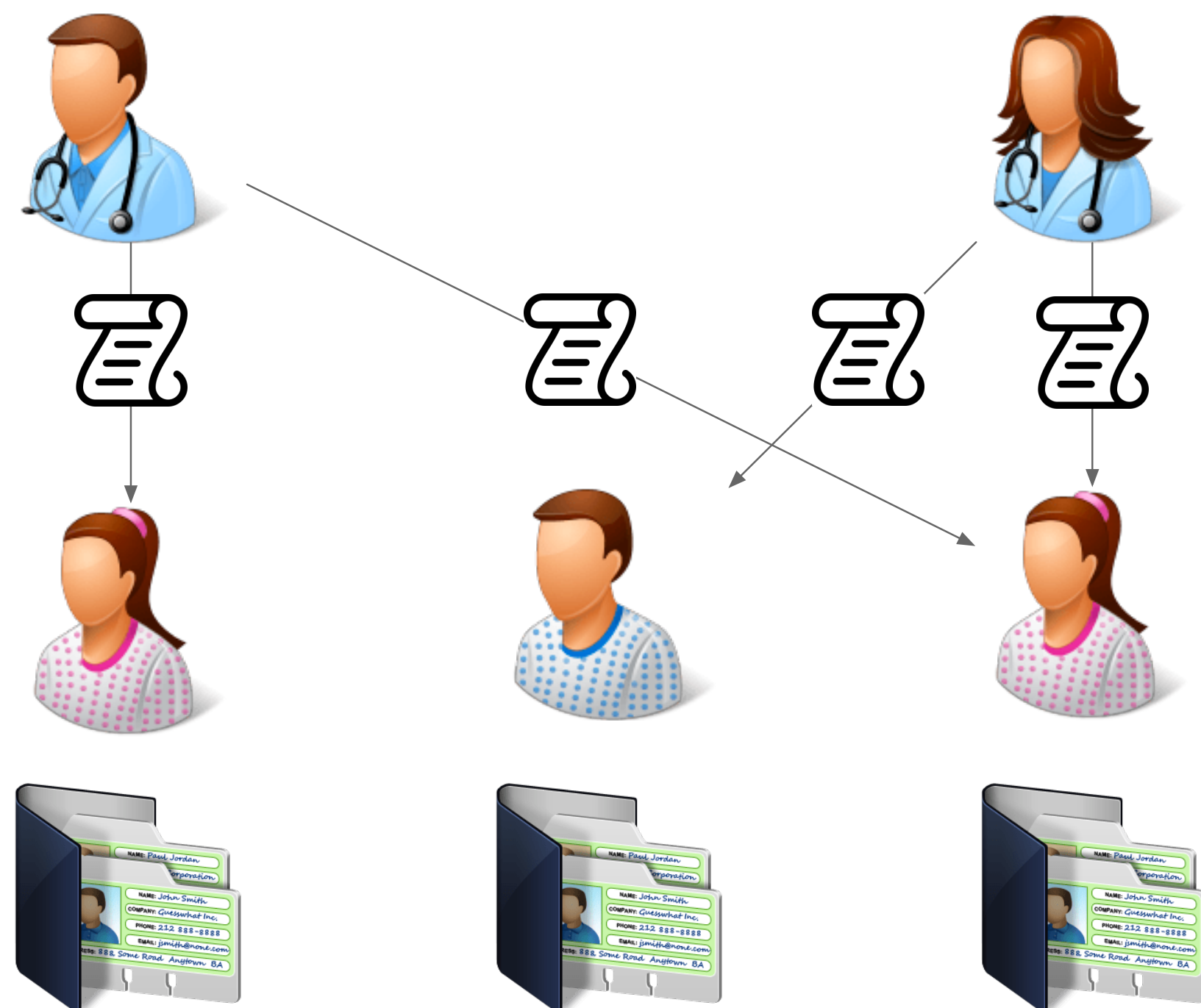
```
public interface PatientLog {  
    void add(String diagnosis);           // Add new diagnosis for today  
    void add(Calendar date, String diagnosis); // Add (old?) diagnosis for given date  
}
```

We don't care about the implementation (List-based? Map-based? Magic-based?)

Our hospital system: The bigger picture



Our hospital system:



Our hospital system: Custom Format

- **Pros:**

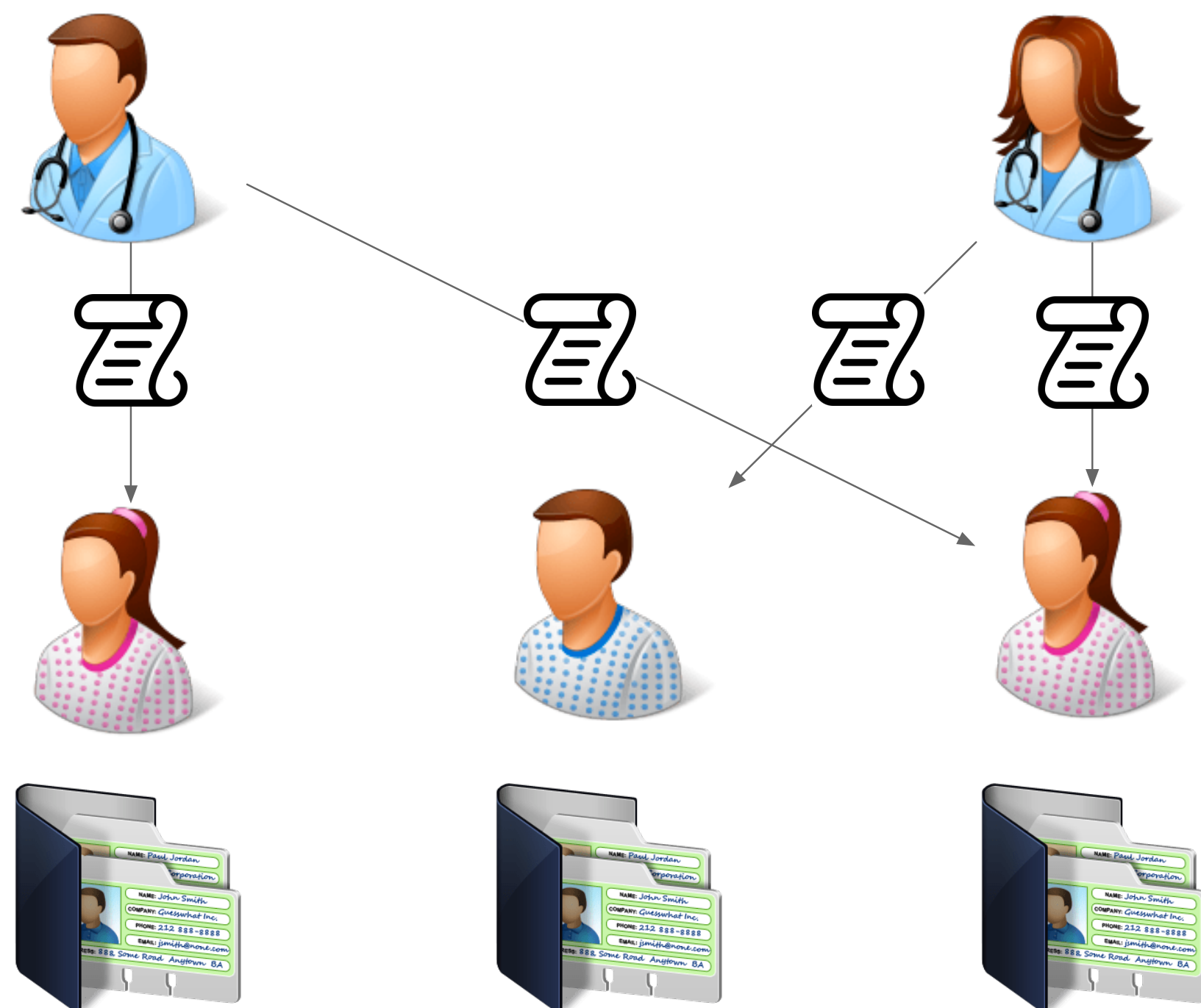
- Human-readable and editable data.

- **Cons:**

- Time-consuming and error-prone to implement.
- Requires writing custom parsing and formatting functions.



Our hospital system:



Our hospital system: Java Serialization

- **Pros:**

- Effortless data storage in Java.
- Minimal coding required.

- **Cons:**

- Not human-readable or easily processed by non-Java applications.
- Tightly coupled to Java, limiting interoperability.
- Watch out with reading/writing data of old versions.



Our hospital system: JSON Serialization (or XML, YAML, ...)

•Pros:

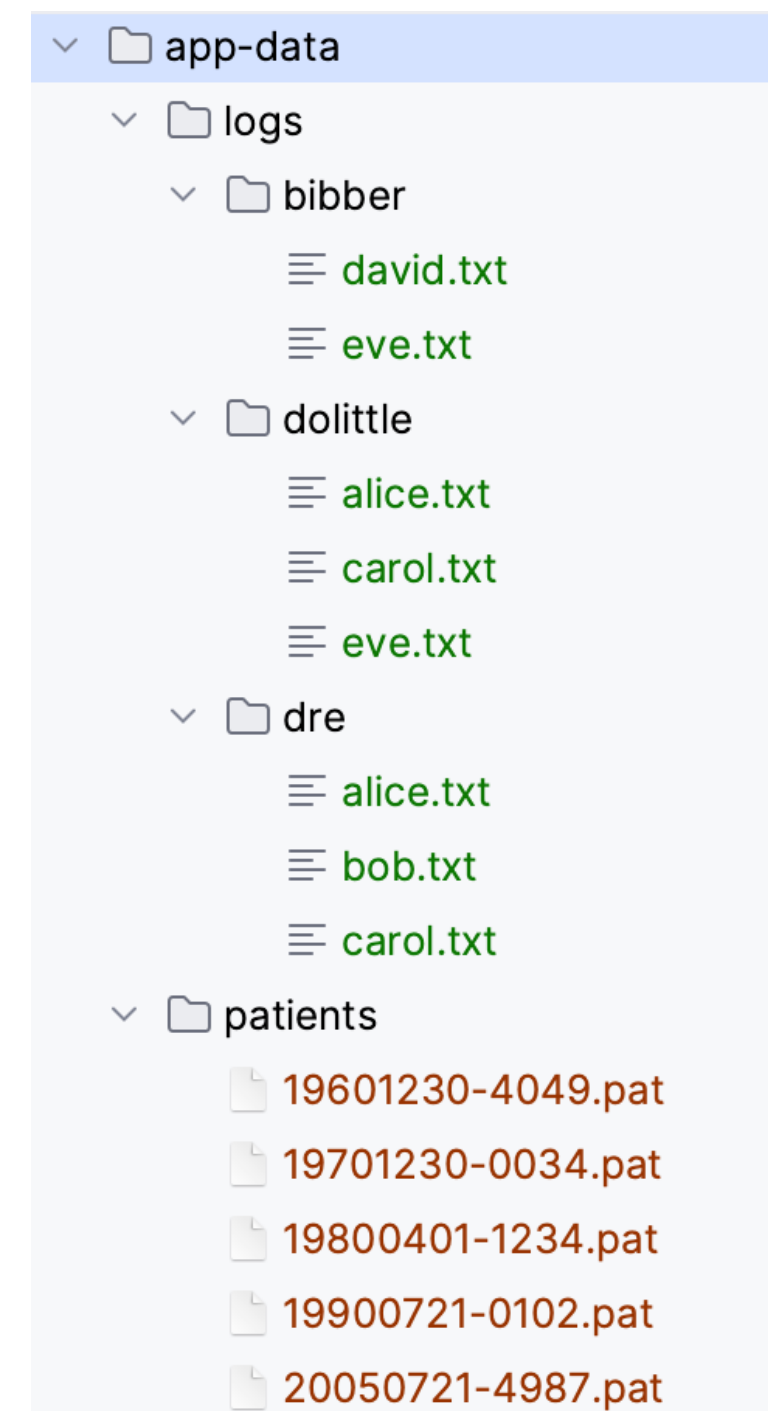
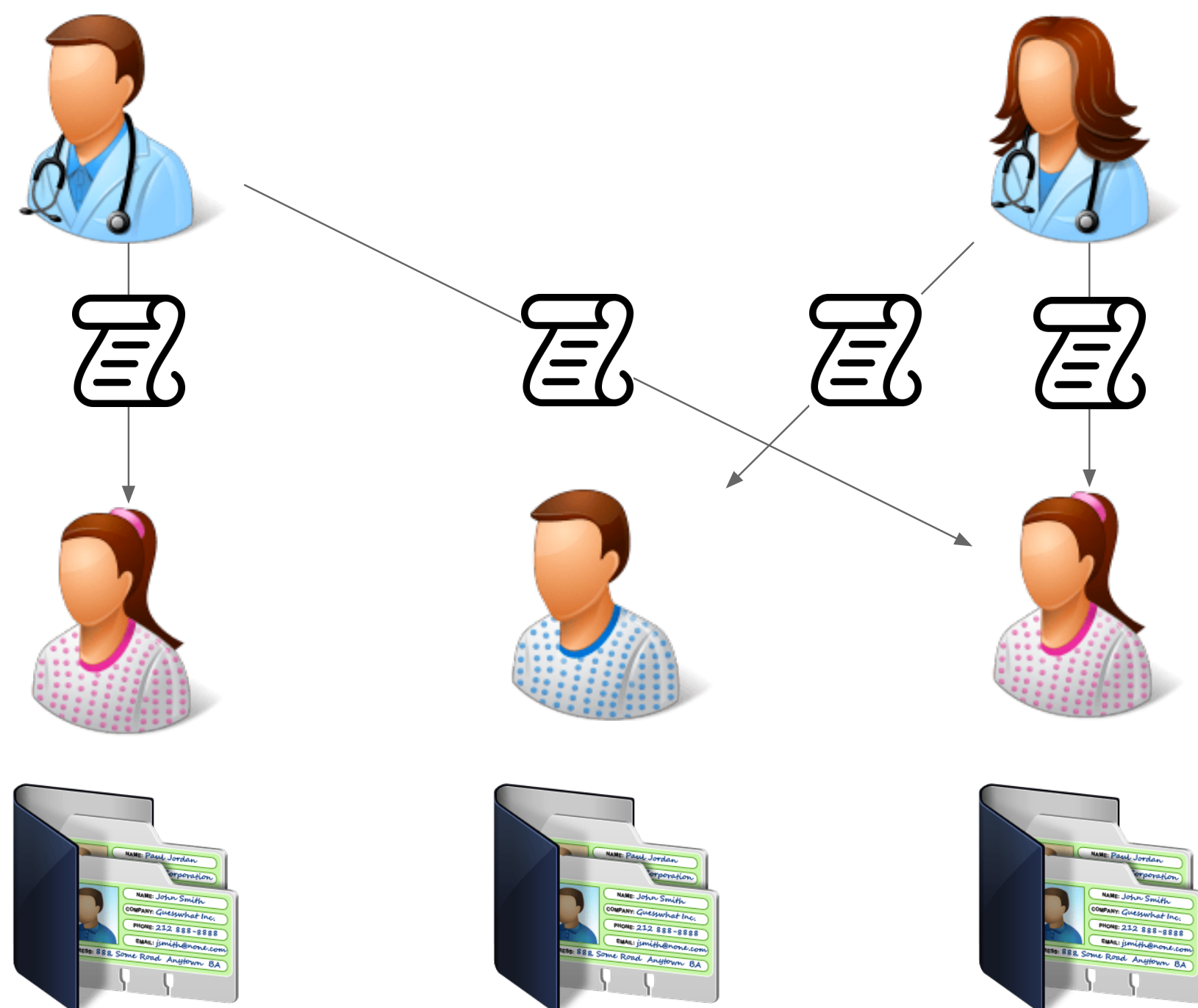
- Relatively easy to program using established libraries.
- Data is readable/writable for informed humans.
- Interoperable with other programs, including different languages.

•Cons:

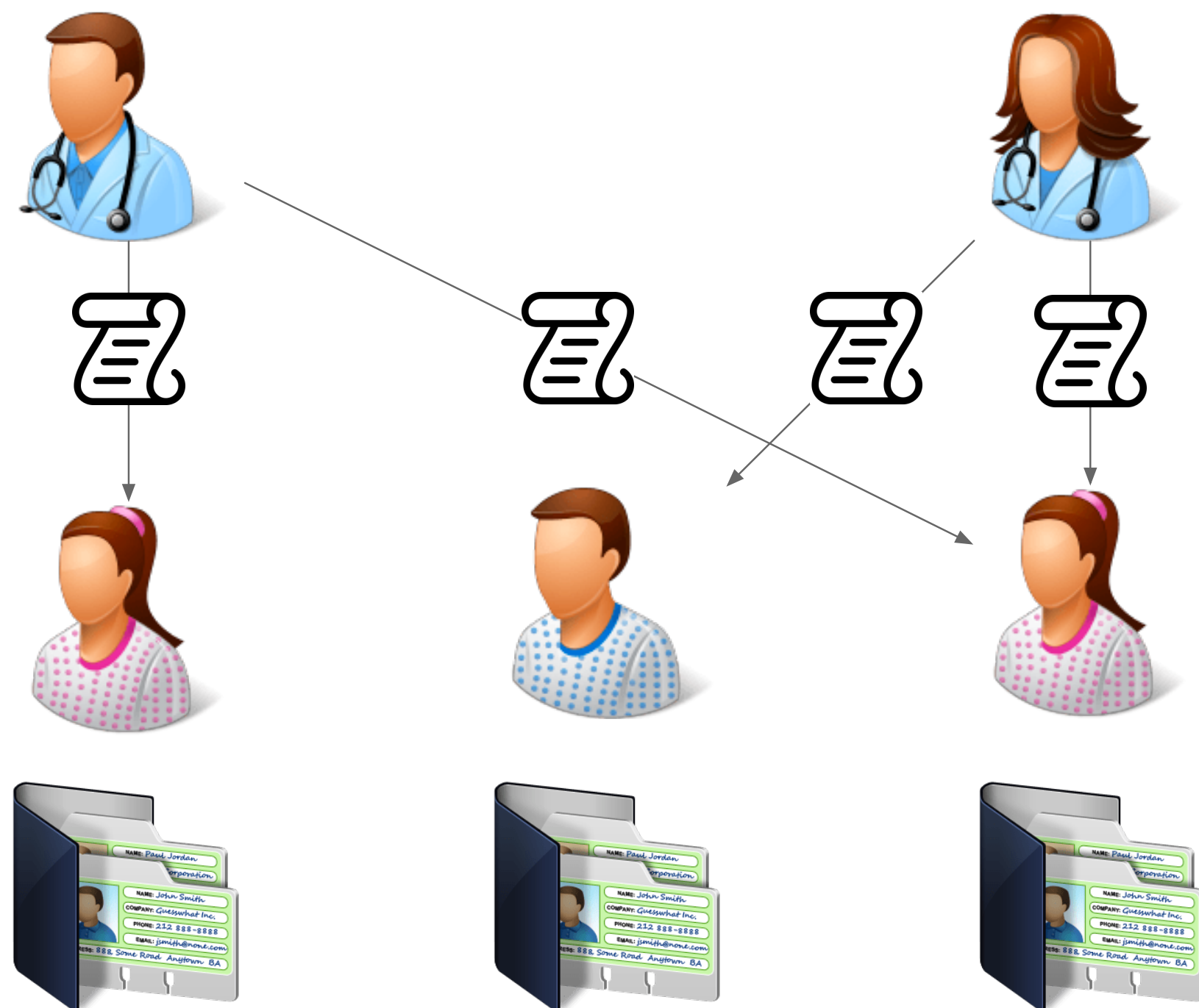
- Slightly more complex than Java Serialization.
- Requires library dependencies for serialization/deserialization.



Our hospital system: More than one file per patient?



Our hospital system: What do we need (technically)?



Read/write text to command line.

Read/write text to file.

Read/write objects to file.

Read/write directories/files.

Serialize objects to JSON.



Object Oriented Architectures and Secure Development

Read/write text to command line.

Arne Debou

Mattias De Wael

Frédéric Vlummens

Standard IO

```
Scanner in = new Scanner(System.in);  
System.out.println("What is your name?");  
String name = in.nextLine();  
System.out.println("Hello " + name);
```

```
What is your name?  
>
```

Standard IO

```
Scanner in = new Scanner(System.in);  
System.out.println("What is your name?");  
String name = in.nextLine();  
System.out.println("Hello " + name);
```

```
What is your name?  
> Alice
```


Standard IO

```
Scanner in = new Scanner(System.in);  
System.out.println("What is your name?");  
String name = in.nextLine();  
System.out.println("Hello " + name);
```

```
What is your name?  
> Alice  
Hello Alice
```

Standard IO

```
Scanner in = new Scanner(System.in);  
System.out.println("What is your name?");  
String name = in.nextLine();  
System.out.println("Hello " + name);
```

What are `System.in` and `System.out`?

PrintStream

```
System.out.println("Hello World");
```

- This is plain Java.
- Classes, objects, and methods.



PrintStream

```
System.out.println("Hello World");
```

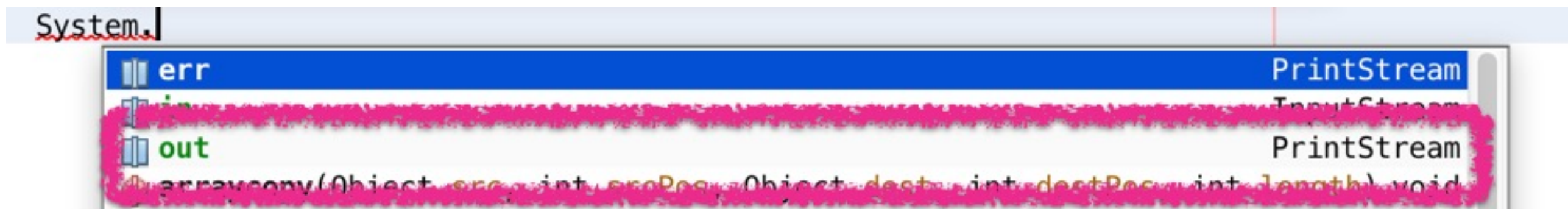
class

field
(object)

method

argument
(String)

All objects have a type



```
PrintStream ps = System.out;  
ps.println("Hello World");
```



PrintStream

```
public void foo( PrintStream ps ) {  
    ps.println("Hello World");  
}
```

```
public void bar() {  
    foo( System.out );  
}
```

PrintStream to Standard Output

```
private void run() {  
    Product p1 = new Product("smartphone", 599);  
    printProduct(System.out, p1);  
}  
  
private void printProduct(PrintStream ps, Product p)  
{  
    ps.printf("Name: \t%s\n", p.getName());  
    ps.printf("Price: \t%.2f\n", p.getPrice());  
}
```

PrintStream to Error

```
private void run() {  
    Product p1 = new Product("smartphone", 599);  
    printProduct(System.err, p1);  
}
```

```
private void printProduct(PrintStream ps, Product p)  
{  
    ps.printf("Name: \t%s\n", p.getName());  
    ps.printf("Price: \t%.2f\n", p.getPrice());  
}
```

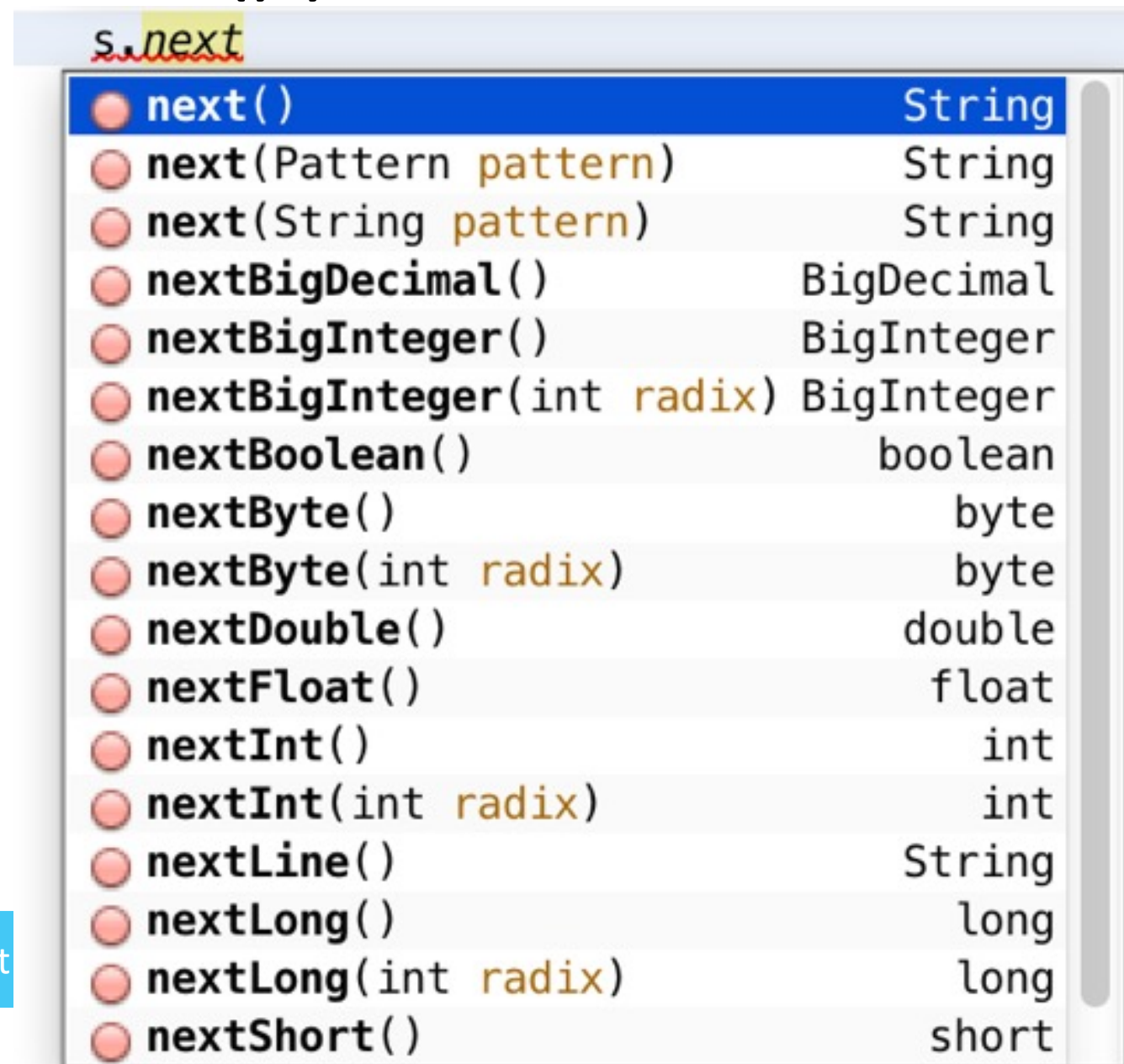
Printf and String.format *(need more control? Search the internet!)*

```
System.err.printf("Name: \t%s\nPrice: \t%.2f\n",  
    p.getName(),  
    p.getPrice()  
);
```

%s	any	String, or implicitly object.toString()
%d	Simple number	[byte, short, int, long]
%f	Decimal Number	[float, double]
%4d	Simple Number	Occupies 4 positions
%02d	Simple Number	Occupies 2 position, pad with zero's.
%-20s	any	Occupies 20 positions, pad with space, - → align left (right is default)
%.2f	Decimal Number	Always 2 numbers after the decimal point
%n	none	OS independent newline



























Scanner

```
Scanner in = new Scanner( System.in );  
while( in.hasNextLine() ) {  
    System.out.println( in.nextLine() );  
}
```



Scanner

```
Scanner in = new Scanner( System.in );  
while( in.hasNextLine() ) {  
    System.out.println( in.nextLine() );  
}
```

Choose Declaration	
 	Scanner(Readable, Pattern) (of java.util.Scanner)
 	Scanner(Readable) (of java.util.Scanner)
 	Scanner(InputStream) (of java.util.Scanner)
 	Scanner(InputStream, String) (of java.util.Scanner)
 	Scanner(InputStream, Charset) (of java.util.Scanner)
 	Scanner(File) (of java.util.Scanner)
 	Scanner(File, String) (of java.util.Scanner)
 	Scanner(File, Charset) (of java.util.Scanner)
 	Scanner(File, CharsetDecoder) (of java.util.Scanner)
 	Scanner(Path) (of java.util.Scanner)
 	Scanner(Path, String) (of java.util.Scanner)
 	Scanner(Path, Charset) (of java.util.Scanner)
 	Scanner(String) (of java.util.Scanner)

Standard IO

Both `System.in` and `System.out` are IO-streams.

(not to be confused with the stream-API of map filter and reduce!)

They represent the command-line from the System. If we can link them to the data of `a file` we can replace Standard IO with file IO.



Object Oriented Architectures and Secure Development

Read/write text to file.

Arne Debou

Mattias De Wael

Frédéric Vlummens

PrintStream to File

```
private void run() {  
    Product p1 = new Product("smartphone", 599);  
    printProduct(System.err, p1);  
  
    printProduct(new PrintStream("product.txt"), p1);  
    printProduct(new PrintStream("/product.txt"), p1);  
    printProduct(new PrintStream("./product.txt"), p1);  
}  
  
private void printProduct(PrintStream ps, Product p) {  
    ps.printf("Name:\t%s\n", p.getName());  
    ps.printf("Price:\t%.2f\n", p.getPrice());  
}
```

Figure out where Java stores the file on your HD.

PrintStream to File

```
String fileName = "test.txt";  
PrintStream ps = new PrintStream("test.txt");
```

```
public PrintStream(String fileName) throws FileNotFoundException {  
    this(false, new FileOutputStream(fileName));  
}
```

```
public FileOutputStream(String name) throws FileNotFoundException {  
    this(name != null ? new File(name) : null, false);  
}
```

PrintStream to File

```
PrintStream ps = new PrintStream(  
    "test.txt"  
);
```

Just the file name ...

PrintStream to File

```
PrintStream ps = new PrintStream(  
    new FileOutputStream(  
        "test.txt"  
    )  
);
```

New file outputstream (with filename)

PrintStream to File

```
PrintStream ps = new PrintStream(  
    new FileOutputStream(  
        new File("test.txt")  
    )  
);
```

Pass a file to the file outputstream

PrintStream to File

```
PrintStream ps = new PrintStream(  
    new FileOutputStream(  
        new File("test.txt")  
    )  
);
```

Different versions, to give the developer more or less control (default vs configuration).

PrintStream to File

```
PrintStream ps = new PrintStream(  
    new FileOutputStream(  
        new File("test.txt")  
    )  
);
```

Create: add data to file, if it exists, clear the file first.

PrintStream to File

```
PrintStream ps = new PrintStream(  
    new FileOutputStream(  
        "test.txt"  
    )  
);
```

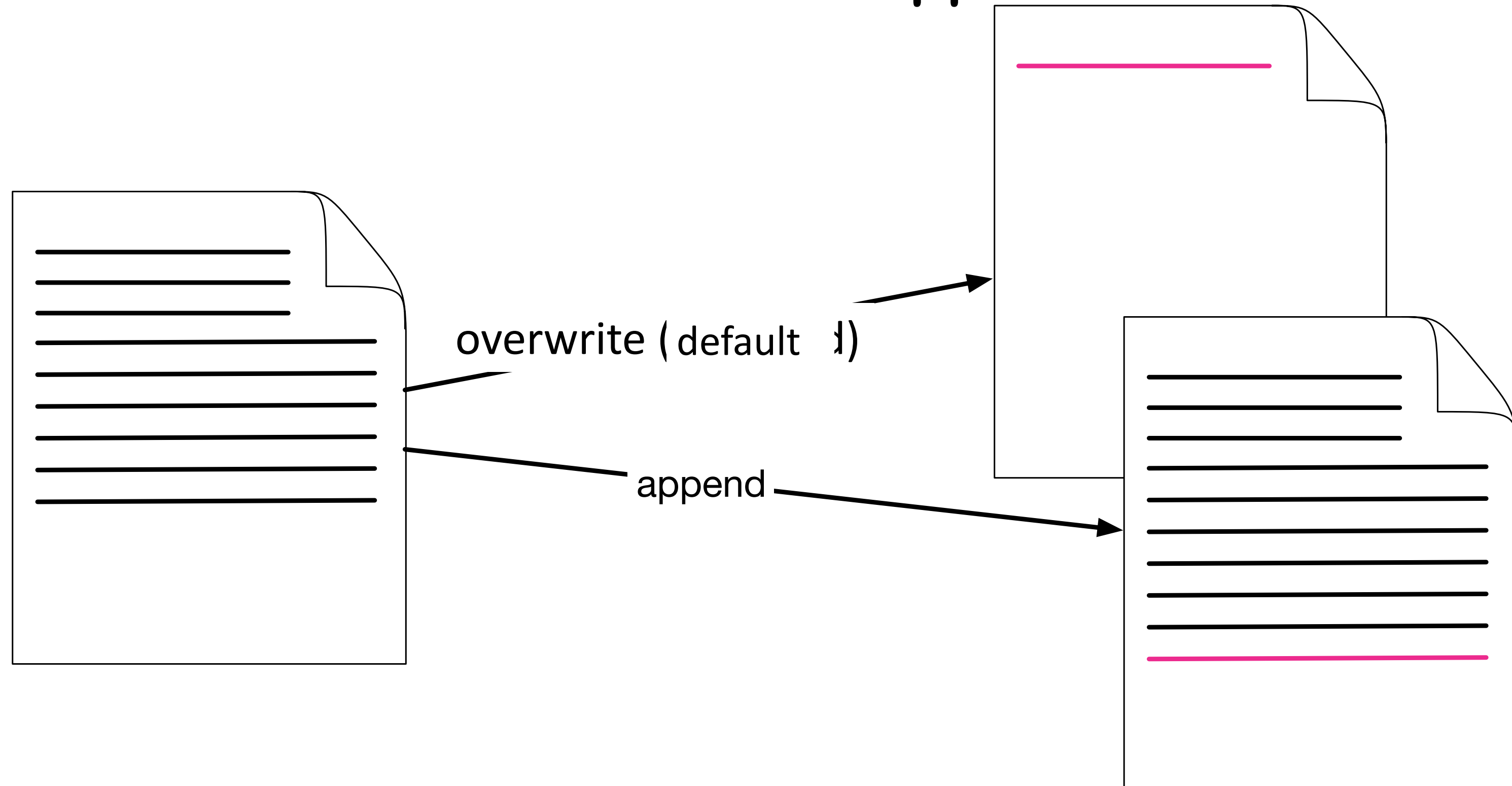
Create: add data to file, if it exists, clear the file first.

PrintStream to File: append

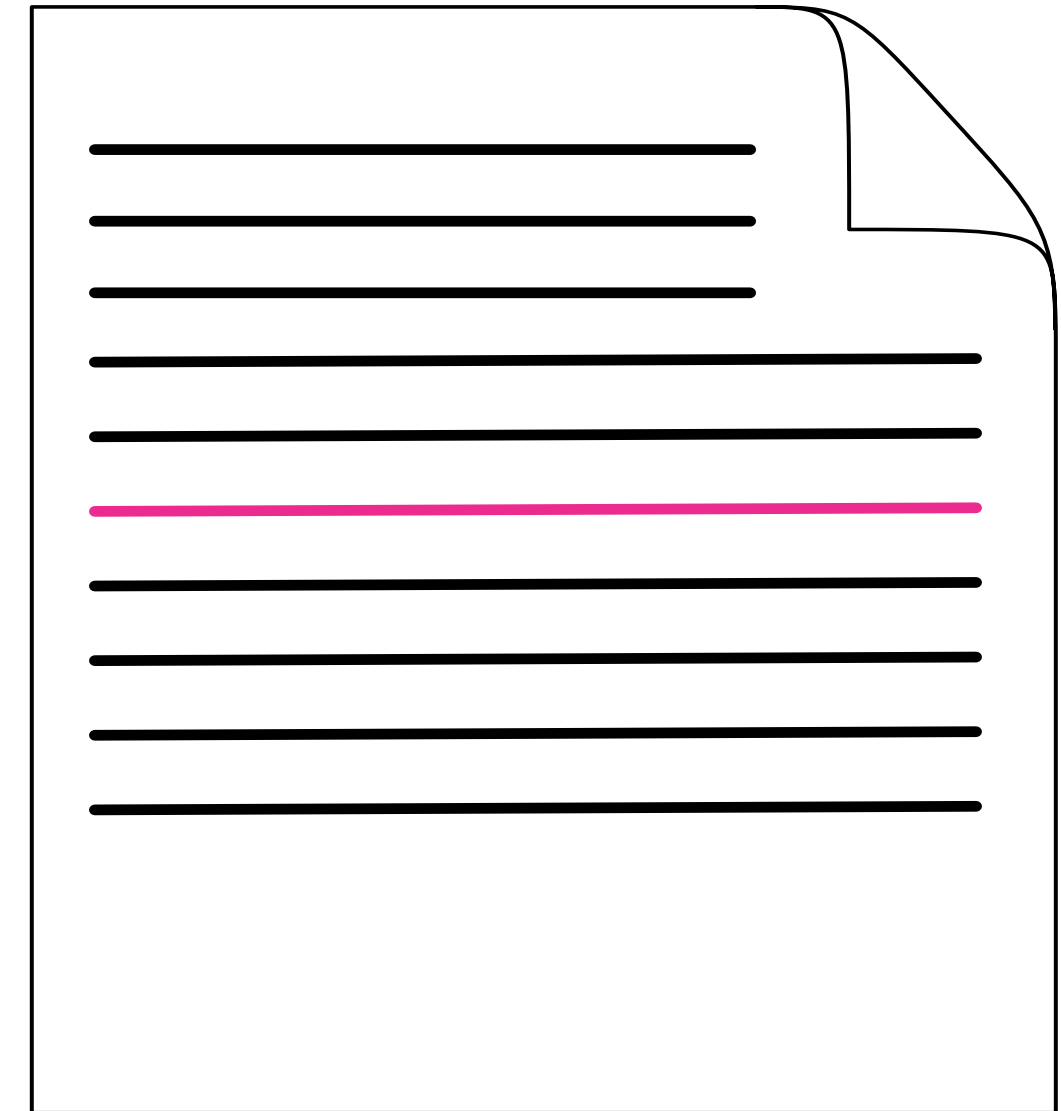
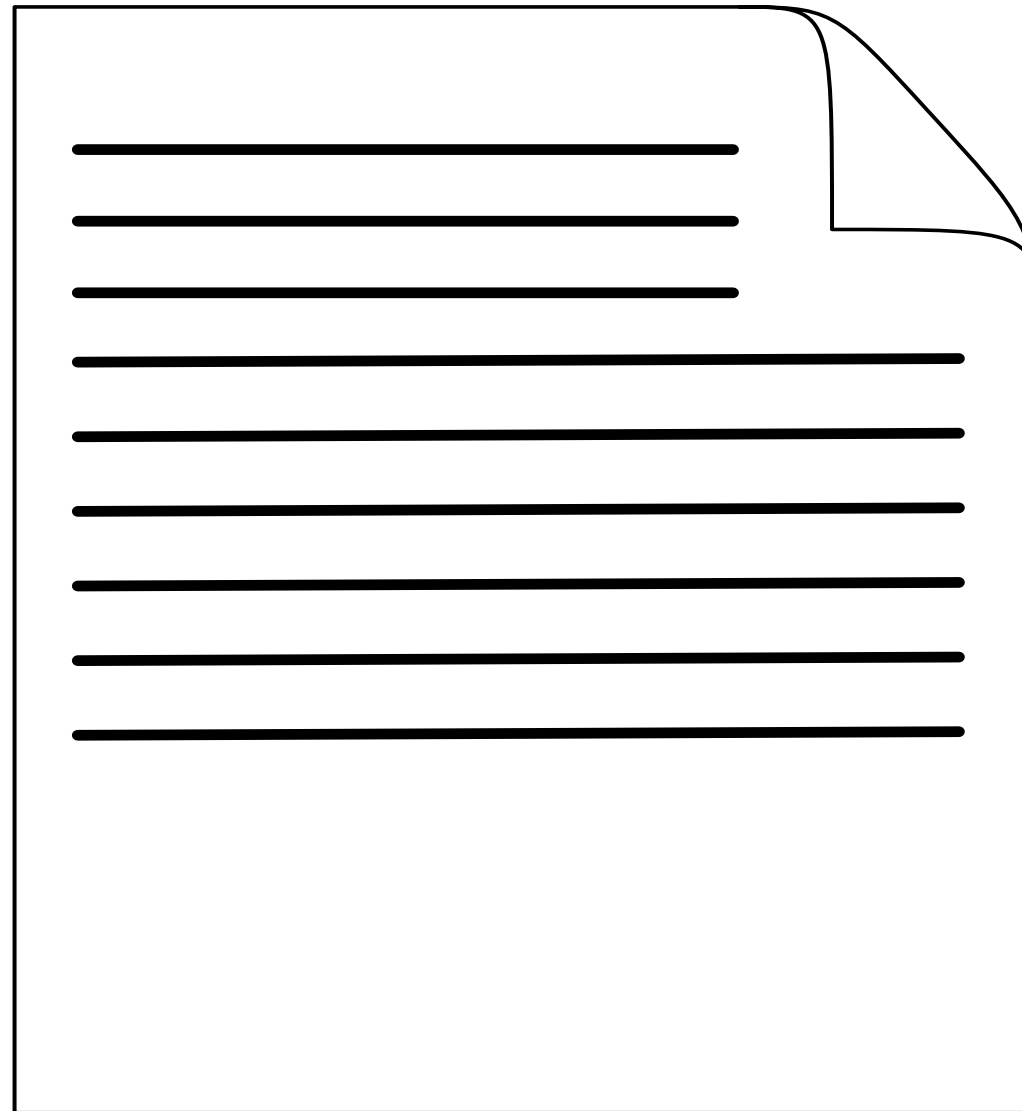
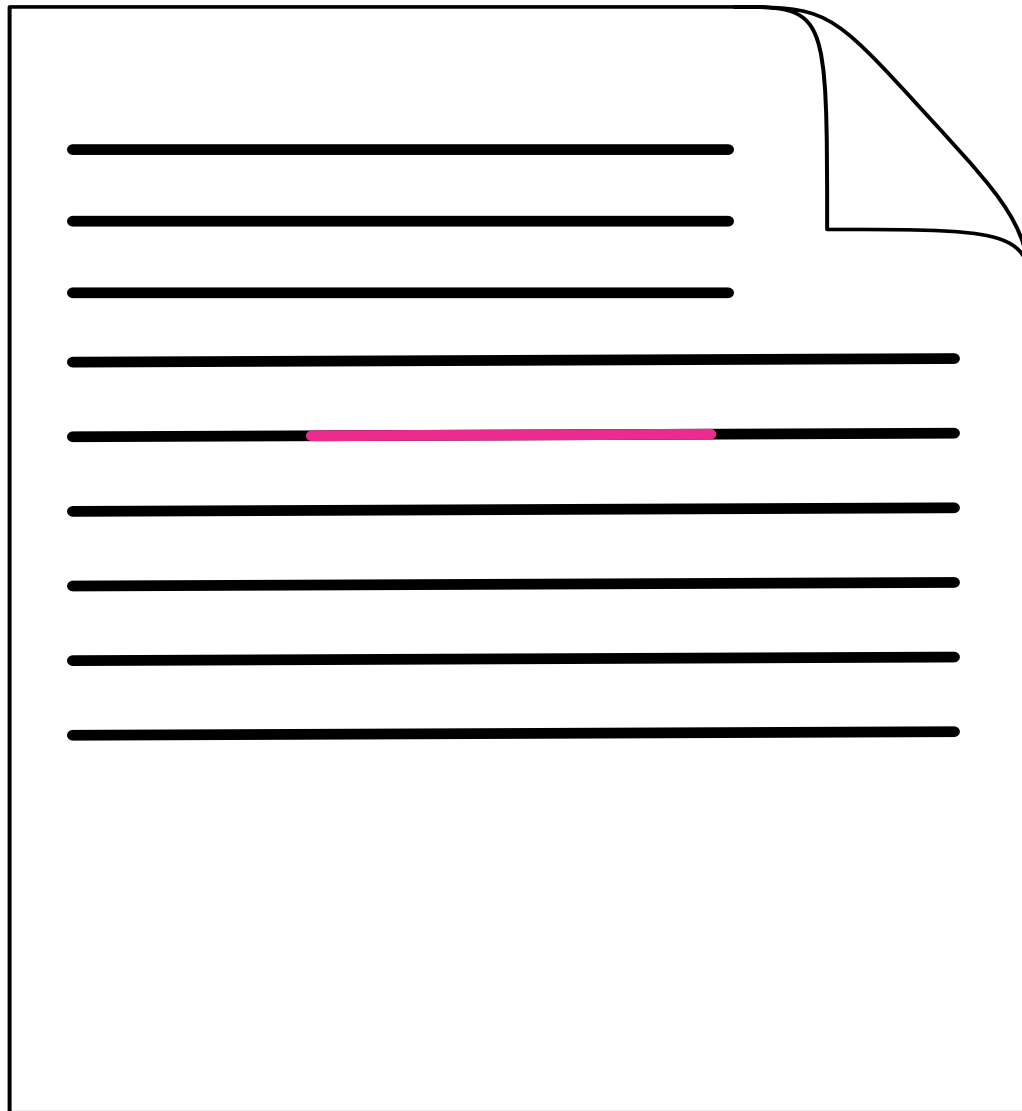
```
PrintStream ps = new PrintStream(  
    new FileOutputStream(  
        "test.txt", true  
    )  
);
```

Append: default is false, when true data is added at the end of the file instead of “clearing” the file first.

PrintStream to File: create versus append



Random access



We do not study random access in files

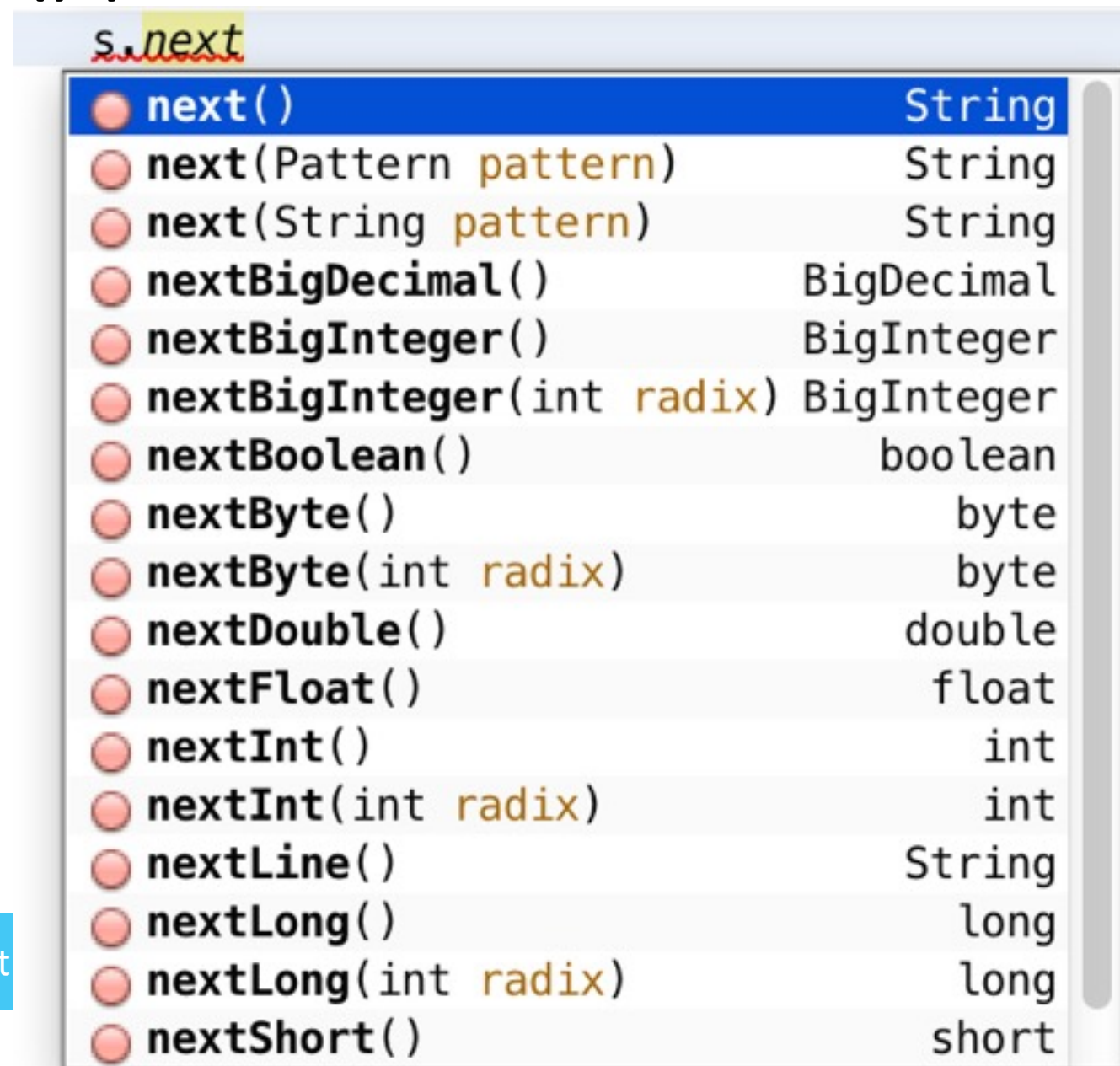
<https://imgtfy.com/?q=java+file+random+access>

Scanner from File

```
Scanner s = new Scanner( new File("test.txt") );  
while( s.hasNext() ) {  
    System.out.println( s.next() );  
}
```

`s.hasNextLine()`

`s.nextLine()`



IO operations tend to go wrong sometimes ...

```
try {  
    Scanner s = new Scanner( new File("test.txt") );  
    while( s.hasNext() ) {  
        System.out.println(s.next());  
    }  
} catch (FileNotFoundException e) {  
    // handle it !  
}
```

IO operations tend to go wrong sometimes ...

```
try {  
    PrintStream ps = new PrintStream(new FileOutputStream(  
        new File("test.txt")  
    ));  
    ps.println("I have so much content!");  
} catch (FileNotFoundException e) {  
    // handle it !  
}
```

IO operations tend to allocate resources (close!!!) ...

```
try {  
    Scanner s = new Scanner( new File("test.txt") );  
    while( s.hasNext() ) {  
        System.out.println(s.next());  
    }  
  
    s.close(); // NOT SAFE (enough) !!!  
} catch (FileNotFoundException e) {  
    // handle it !  
}
```

IO operations tend to allocate resources (close!!!) ...

```
try {  
    PrintStream ps = new PrintStream(new FileOutputStream(  
        new File("test.txt")  
    ));  
    ps.println("I have so much content!");  
    ps.close(); // NOT SAFE (enough) !!!  
} catch (FileNotFoundException e) {  
    // handle it !  
}
```

IO operations tend to allocate resources (close!!!) ...

```
try (Scanner s = new Scanner( new File("test.txt") )) {  
    while( s.hasNext() ) {  
        System.out.println(s.next());  
    }  
} catch (FileNotFoundException e) {  
    // handle it !  
}
```

IO operations tend to allocate resources (close!!!) ...

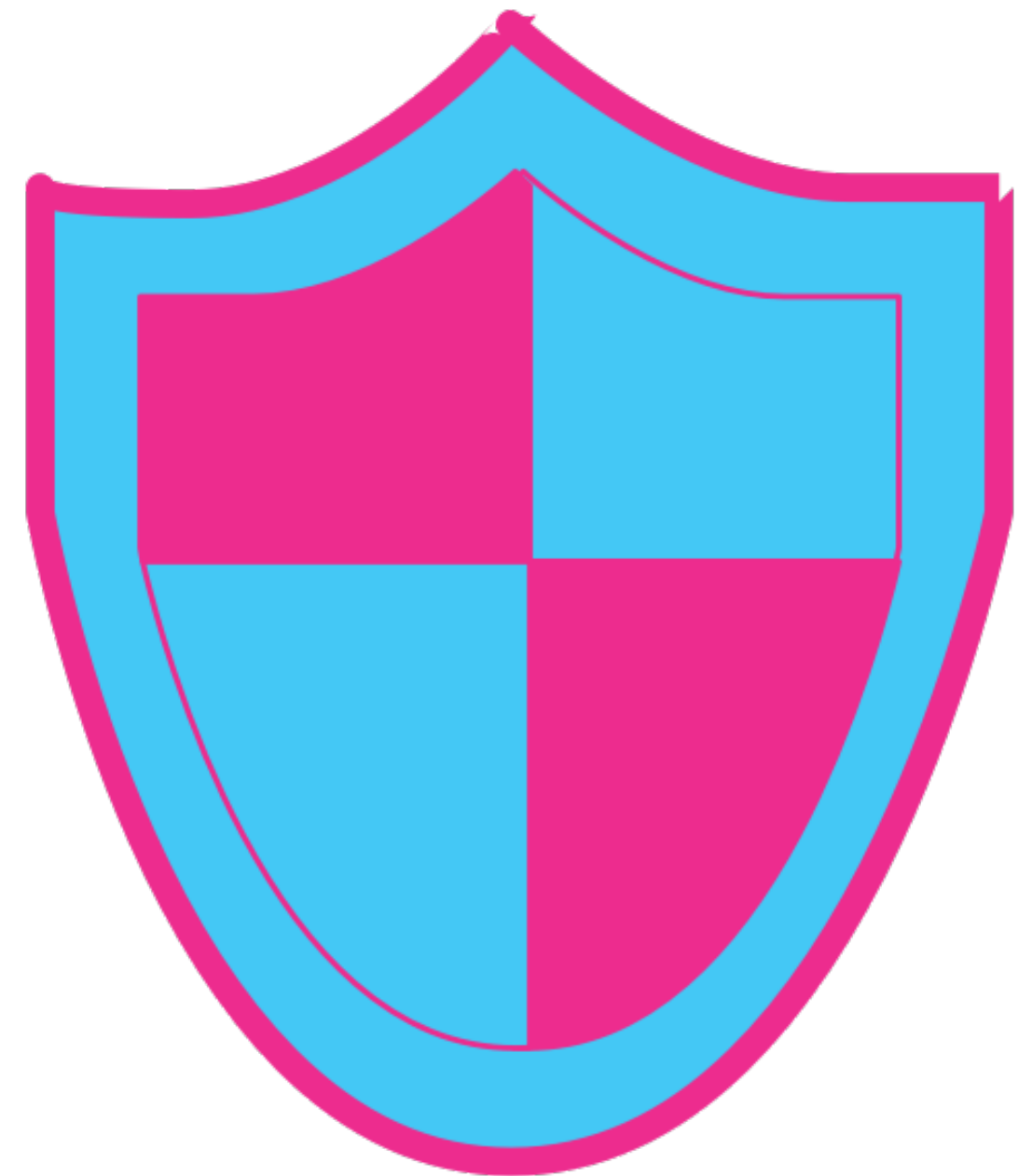
```
try (  
    PrintStream ps = new PrintStream(new FileOutputStream(  
        new File("test.txt")  
    )  
)) {  
    ps.println("I have so much content!");  
} catch (FileNotFoundException e) {  
    // handle it !  
}
```

Guideline 1-2: Release resources in all cases

<https://www.oracle.com/technetwork/java/seccodeguide-139067.html#1-2>

```
try ( /* put your resources here */ ) {  
    /* put your computations here */  
} catch (FileNotFoundException e) {  
    // handle it !  
}
```

Some objects, such as open files, locks and manually allocated memory, behave as resources which require every acquire operation to be paired with a definite release. It is easy to overlook the vast possibilities for executions paths when exceptions are thrown. Resources should always be released promptly no matter what.

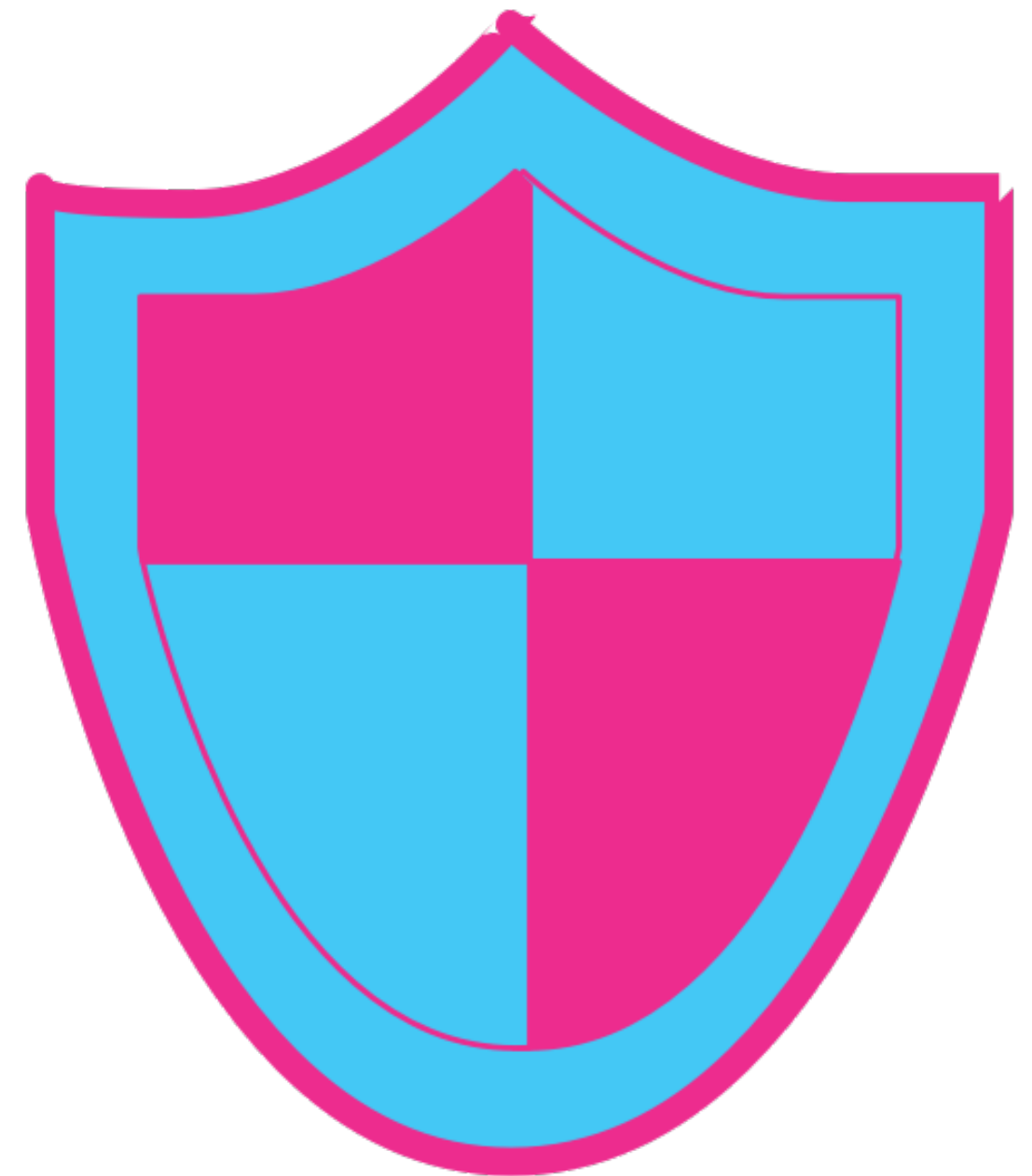


Guideline 5-1 / INPUT-1: Validate inputs

<https://www.oracle.com/java/technologies/javase/seccodeguide.html#5-1>

Input from untrusted sources must be validated before use. Maliciously crafted inputs may cause problems, whether coming through method arguments or external streams. Examples include overflow of integer values and directory traversal attacks by including "../" sequences in filenames. Ease-of-use features should be separated from programmatic interfaces.

The content of the files is written by our program,
It should be trusted. However, we cannot guarantee that
nobody tampered with the files...





Object Oriented Architectures and Secure Development

Read/write objects to file.

Arne Debou

Mattias De Wael

Frédéric Vlummens

Write objects ...

```
try (  
    ObjectOutputStream out = new ObjectOutputStream(  
        new FileOutputStream("./path/to/file.ext")  
    ) {  
        out.writeObject(patient);  
    } catch(IOException e) {  
        ...  
    }
```

Read objects ...

```
try (  
    ObjectInputStream in = new ObjectInputStream(  
        new FileInputStream ("../path/to/file.ext"))  
    ){  
    patients.add((Patient)in.readObject());  
} catch (IOException | ClassNotFoundException e) {  
    ...  
}
```

Cast to patient: *WE know the file contains a patient, but Java cannot. This is one of the rare cases where the programmers know more about types than Java.*

Serializable

This only works if the class you read/write is **Serializable**.

A class is **Serializable** if:

- the class **implements Serializable**;
- all the fields are **Serializable**.

By default, many Java built-in classes are serializable by default:

- primitive types,
- most collections (implementations).

De-Serializable

If you want your class to be deserializable as well, make sure the super-class is also serializable,

The class of the object you are reading (deserialize) and the current class are **the same version!**

You can make the latter explicit by providing a *serialVersionUID*:

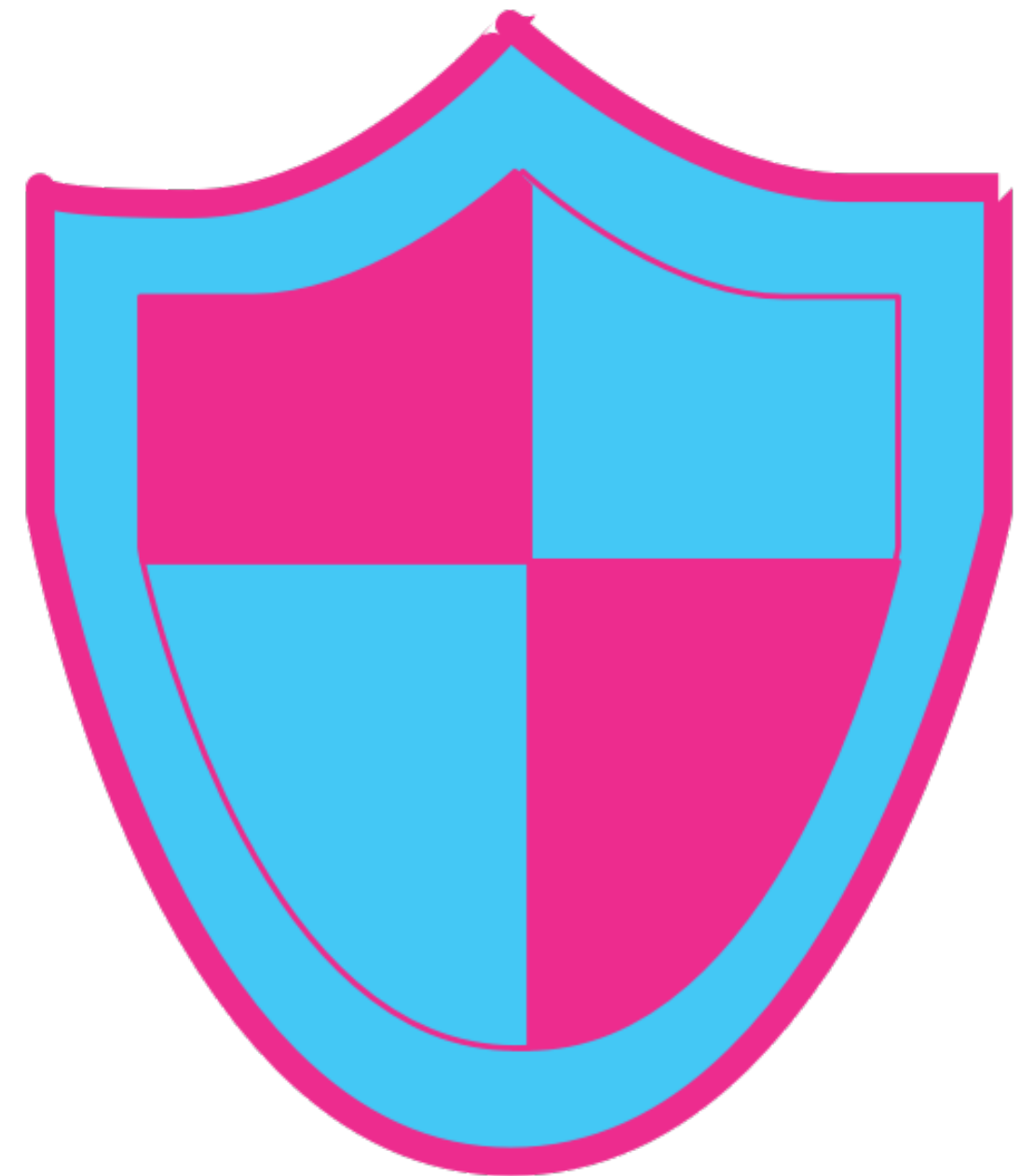
@Serial

private static final long *serialVersionUID* = 1L;

Guideline 1-2: Release resources in all cases

<https://www.oracle.com/technetwork/java/seccodeguide-139067.html#1-2>

```
try ( /* put your resources here */ ) {  
    /* put your computations here */  
} catch (FileNotFoundException e) {  
    // handle it !  
}  
    Holds for Serializing as well (if to file or other resource)
```



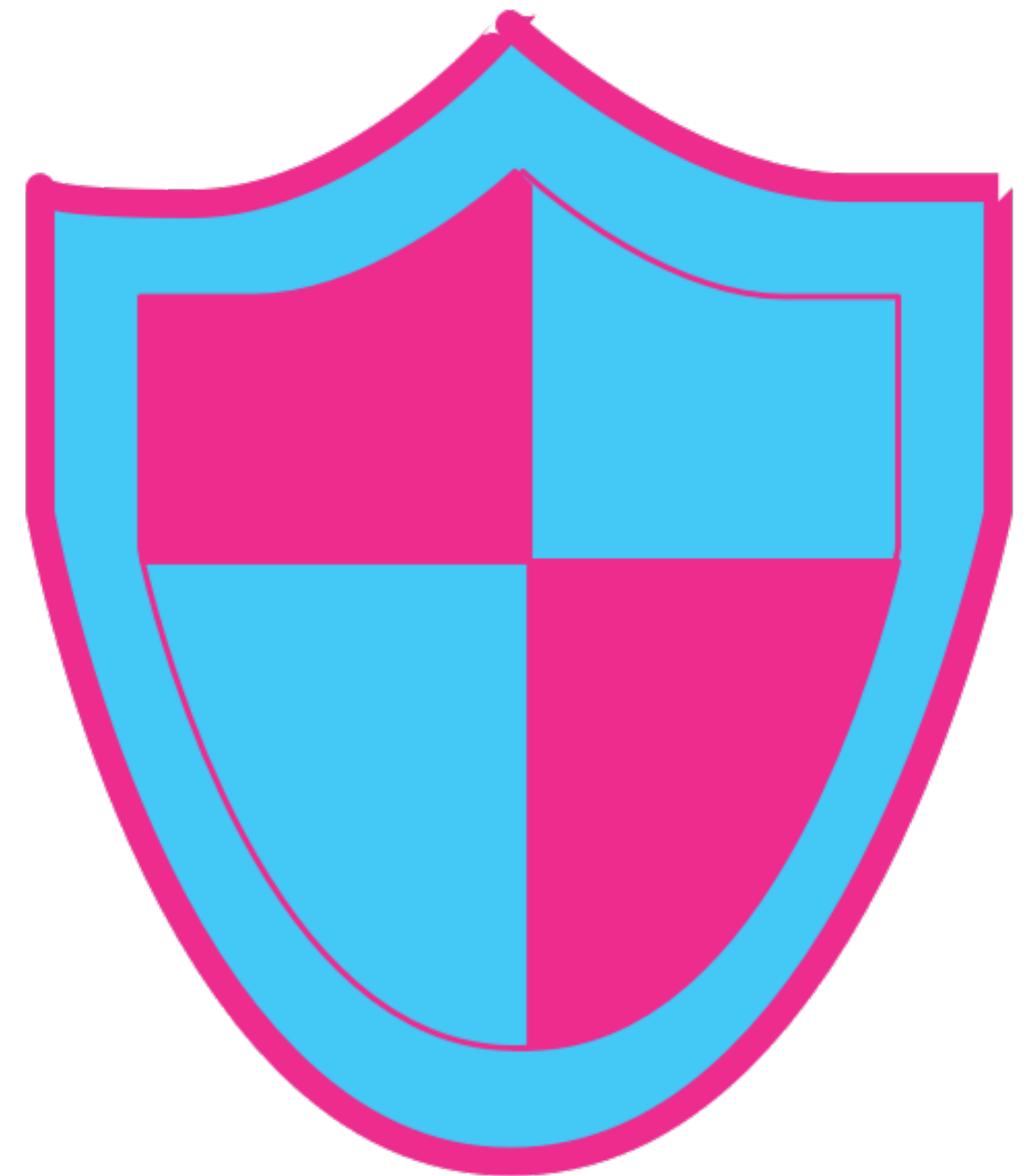
Guideline 8-1 / SERIAL-6 through Guideline 8-6 / SERIAL-6:

<https://www.oracle.com/java/technologies/javase/seccodeguide.html#8>

Deserialization of untrusted data is inherently dangerous and should be avoided.

There is a whole 'chapter' about Serialization/Deserialization in the Secure Coding Guidelines ...

See your courses on OSs and networks to store your files somewhere safe.





Object Oriented Architectures and Secure Development

Read/write directories/files.

Arne Debou

Mattias De Wael

Frédéric Vlummens

File-class

File is a wrapper for a path-to-file-or-dir, many/most methods are self explanatory:

```
File f = new File("data.txt");
if (!f.exists()) {
    try {
        boolean success = f.createNewFile();
    } catch (IOException e) {
        ...
    }
}
```

File-class

If the File is a directory, we can enumerate its contents:

```
if (d.isDirectory()) {
```

```
}    File d = new File(pathname: "./some/path/with/files");  
    for(File f : d.listFiles()) {  
        doSomething(f);  
    }  
}
```



IntelliJ IDEA autocomplete popup for `File.listFiles()`. The popup shows three methods, all returning `File[]`. The first method, `listFiles()`, is highlighted. The second method is `listFiles(FilenameFilter filter)` and the third is `listFiles(FileFilter filter)`. At the bottom of the popup, it says "Press ← to insert, → to replace".

listFiles()	File[]
listFiles(FilenameFilter filter)	File[]
listFiles(FileFilter filter)	File[]

File-class

If the File is a directory, we can enumerate its contents:

```
File d = new File("./some/path/with/files");
boolean success = d.mkdirs();
for(File f : d.listFiles((dir, fn)->fn.endsWith(".txt"))) {
    doSomethingWith(f);
}
```

The File class

<https://docs.oracle.com/javase/7/docs/api/java/io/File.html>

The File-class represents a file or directory on your hard-disk.

The class contains a lot of handy methods.



Object Oriented Architectures and Secure Development

Serialize objects to JSON.

Arne Debou

Mattias De Wael

Frédéric Vlummens

Jackson (not default part of Java)

Build.gradle: *(watch out with notation if you use Kotlin, i.e., with build.gradle.kts)*

// <https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind>
implementation 'com.fasterxml.jackson.core:jackson-databind:2.15.2'

In code:

```
ObjectMapper json = new ObjectMapper();
```

```
Patient original = new Patient(...);
```

```
String txt = json.writeValueAsString(original);
```

```
Patient deserialized = json.readValue("txt", Patient.class);
```

Jackson: serialize

- uses getters to populate properties (except when annotated with `@JsonIgnore`)

```
public class Person {  
  
    public String getName() {}  
    public Calendar getBirthDate() {}  
    @JsonIgnore public String getSocialSecurityNumber() {}  
    public int computeAge() {}  
  
}  
  
→  
{ "name": "Alice", "birthDate": 1695642693959 }
```

Jackson: serialize

- uses methods/fields annotated with `@JsonProperty` to populate properties.

```
public class Person {  
  
    @JsonProperty("gender") private Gender gender;  
  
    public String getName() {}  
    @JsonIgnore public Calendar getBirthDate() {}  
    @JsonProperty("age") public int computeAge() { /* do computation with birthdate*/ }  
}  
  
→  
  
{"name":"Alice","gender":"F","age":123}
```


Jackson: serialize

More complex situations: consult the documentation or ...

<https://www.baeldung.com/jackson>

Jackson: deserialize

- uses setters in combination with an empty constructor.

Downside: We prefer classes with as few setters as possible and getters for all

Guideline 6-1 / MUTABLE-1: Prefer immutability for value types

Jackson: deserialize

- uses constructor (annotated with `@JsonCreator`) with arguments (annotated with `@JsonProperty`)

```
@JsonCreator
private Person(
    @JsonProperty("name") String name,
    @JsonProperty("age") int age
) {
    this.name = name;
    this.age = age;
}
```

Jackson: deserialize

- uses (static) factory method (annotated with `@JsonCreator`) with arguments (annotated with `@JsonProperty`)

```
@JsonCreator
public static Person build(
    @JsonProperty("name") String name,
    @JsonProperty("age") int age
) {
    return new Person(name, age);
}
```