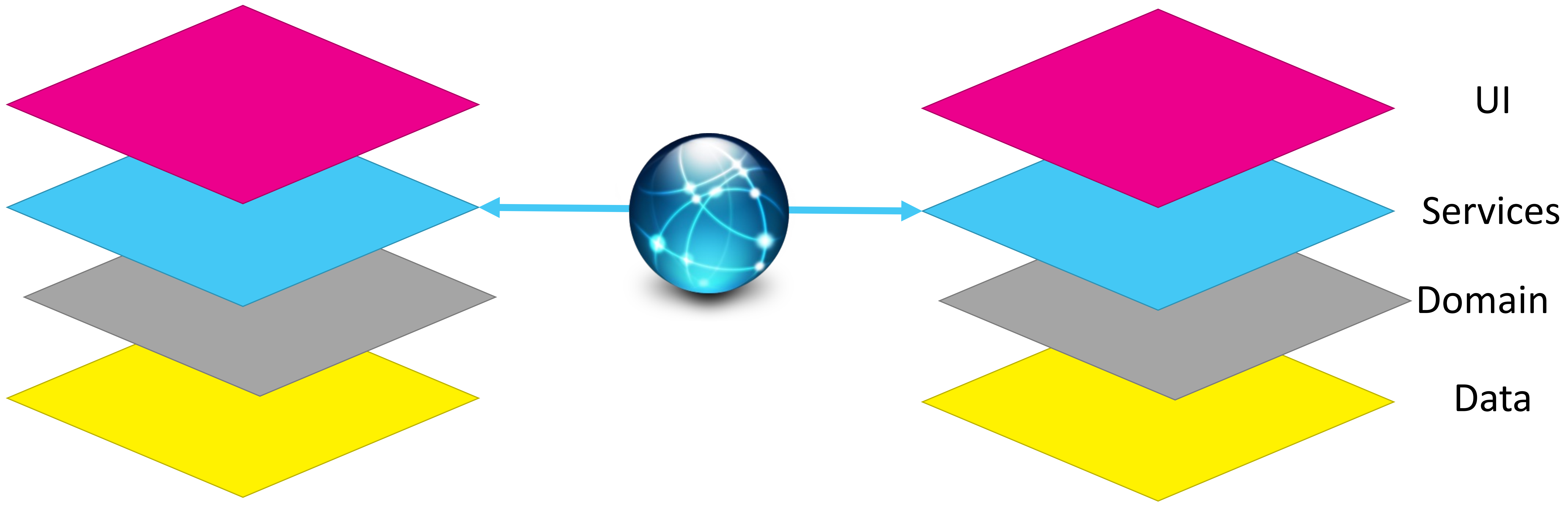
howest
hogeschool

Object Oriented Architectures and Secure Development

Client – Server

Arne Debou
Mattias De Wael
Frédéric Vlummens

Client-server architecture



Communication is done through sockets

Sockets

- Server

```
ServerSocket serverSock = new ServerSocket(1234);
Socket sock = serverSock.accept();
```

Pick a port

Wait for a connection

- Client

```
Socket sock = new Socket("localhost", 1234);
```

The server's address and port

```
sock.getInputStream();
sock.getOutputStream();
```

We know its family (file streams, System.in, System.out, …)

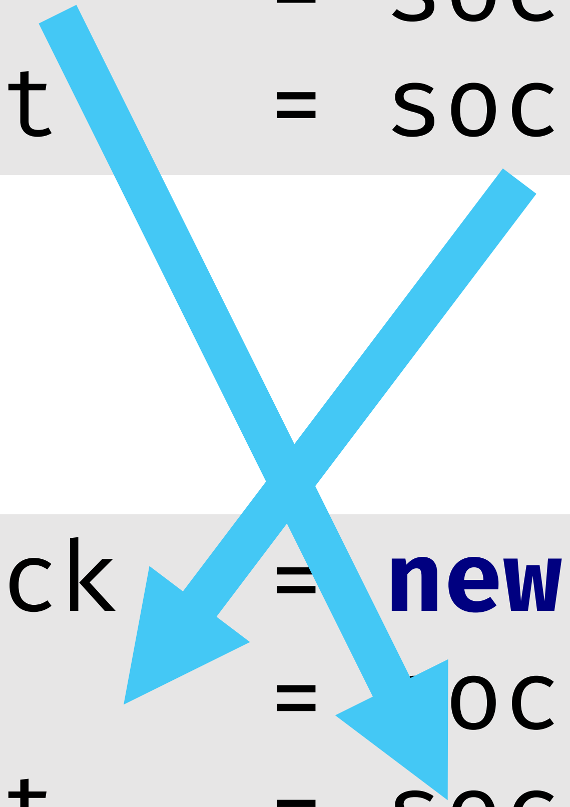
Sockets

- Server

```
Socket       sock   = serverSock.accept();
InputStream  in     = sock.getInputStream();
OutputStream out    = sock.getOutputStream();
```

- Client

```
Socket       sock   = new Socket("localhost", 1234);
InputStream  in     = sock.getInputStream();
OutputStream out    = sock.getOutputStream();
```



Sending and receiving messages over the sockets

```java
Socket sock = new Socket("localhost", 1234);

Scanner in = new Scanner(sock.getInputStream());
PrintStream out = new PrintStream(sock.getOutputStream(), true);
```

Simple example: echo server

```
try (ServerSocket serverSocket = new ServerSocket(1234)) {
    while (true) {
        try (Socket socket = serverSocket.accept()) {
            Scanner in = new Scanner(socket.getInputStream());
            PrintStream out = new PrintStream(socket.getOutputStream());

            while (in.hasNextLine()) {
                String line = in.nextLine();
                out.println(line.toUpperCase());
            }
        }
    }
} catch (IOException ex) {
    …
}
```

SERVER

Simple example: echo server

```
try (Socket socket = new Socket("localhost", 1234)) {
    Scanner in = new Scanner(socket.getInputStream());
    PrintStream out = new PrintStream(socket.getOutputStream());

    Scanner kbd = new Scanner(System.in);      ← keyboard input

    String line = kbd.nextLine();

    while (!line.equals("STOP")) {
        out.println(line);
        String response = in.nextLine();
        System.out.println(response);
        line = kbd.nextLine();
    }
} catch (IOException ex) {
    …
}
```

CLIENT

Client-Server communication

The communication between client and server has to adhere to a specific set of rule:
Simple, like our echo server; or complex like HTTP.

This set of rules is what we call a protocol. (e.g., Hyper Text Transfer Protocol).

Or you use/implement an existing protocol, or you invent one for your own app.
The latter, can not be "explained" in slides, you should try this as an exercise.

Sending and receiving custom objects as messages

- We need to tell Java objects can be converted to byte streams for transmission over the wire…
- Solution: serialization! But we already know about serialization

```
public class Message implements Serializable {
        … <-- anything that is also serializable (see files)
}
```

Pay attention to security!

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

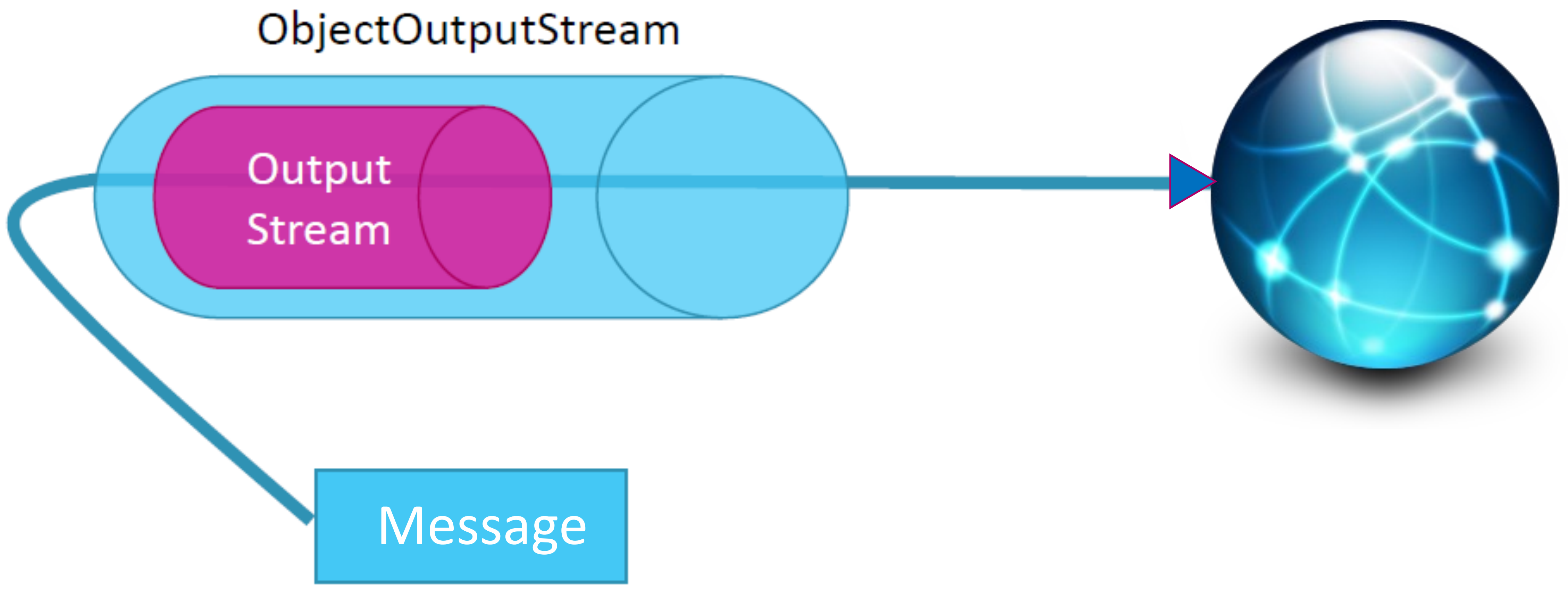
Java Serialization provides an interface to classes that sidesteps the field access control mechanisms of the Java language. As a result, care must be taken when performing serialization and deserialization. Furthermore, deserialization of untrusted data should be avoided whenever possible, and should be performed carefully when it cannot be avoided.

https://www.oracle.com/technetwork/java/seccodeguide-139067.html#8

- Guideline 8-1 / SERIAL-1: Avoid serialization for security-sensitive classes
- Guideline 8-2 / SERIAL-2: Guard sensitive data during serialization
- Guideline 8-3 / SERIAL-3: View deserialization the same as object construction
- Guideline 8-4 / SERIAL-4: Duplicate the SecurityManager checks enforced in a class during serialization and deserialization
- Guideline 8-5 / SERIAL-5: Understand the security permissions given to serialization and deserialization
- **Guideline 8-6 / SERIAL-6: Filter untrusted serial data**

Using ObjectOutputStream and ObjectInputStream

- Java will do the conversion for us if we **decorate** our OutputStreams and InputStreams with ObjectOutputStream and ObjectInputStream, respectively.
- *Decorating is design pattern which "adds new functionality" to an object by wrapping it in another (more powerful) object.*



Server

Message.java

```
public class Message implements Serializable { … }
```

Server.java

```
serverSocket = new ServerSocket(1234);

while (true) {
    try (Socket socket = serverSocket.accept();
         ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
    ) {
        oos.writeObject( new Message(…) );
    } catch (IOException ex) {
        LOGGER.log(Level.WARNING, "Exception during communication with client.", ex);
    }
}
```

Client: just plug in a new implementation of ProductRepository

```
      try (Socket socket = new Socket("localhost", 1234);
         ObjectInputStream ois = new ObjectInputStream(socket.getInputStream())) {
        Message msg = (Message) ois.readObject();
        // do something with message
      } catch (IOException | ClassNotFoundException ex) {
        LOGGER.log(Level.SEVERE, "Unable to read message from network.", ex);
        throw new MySpecialException("Unable to retrieve message.");
      }
    }
}
```

We (not Java) know it is a Message.

When both Client and Server send and read Objects:

```
    try (Socket socket = new Socket("localhost", 1234);
1           ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
2          ObjectInputStream ois = new ObjectInputStream(socket.getInputStream())) {
      …
    } catch (IOException | ClassNotFoundException ex) {
      LOGGER.log(Level.SEVERE, "Unable to read message from network.", ex);
      throw new MySpecialException("Unable to retrieve message.");
    }
  }
}
```

We (not Java) know it is a Message.

When both Client and Server send and read Objects:

make sure to create the object-output stream before the in object-input stream! Else both your applications will block.