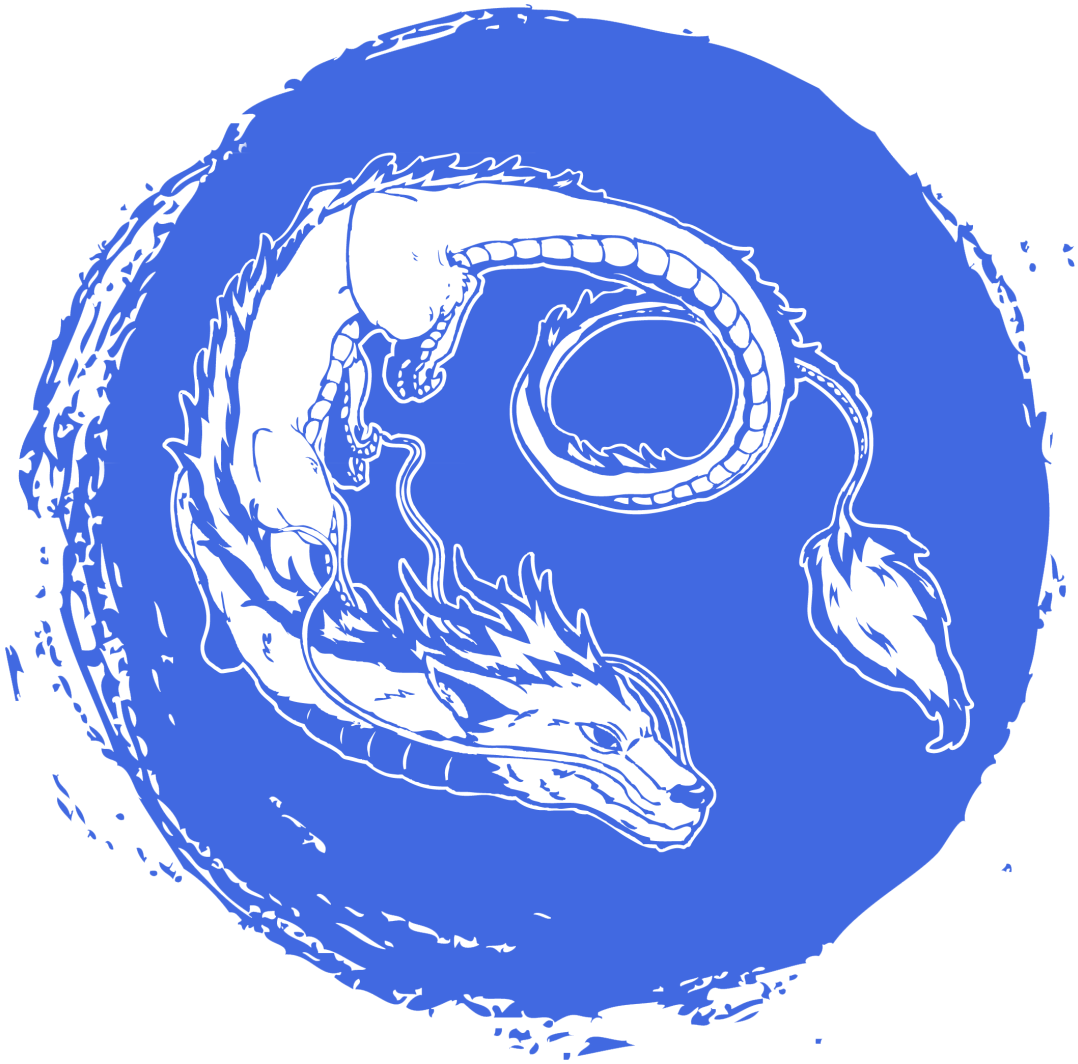

# Java Workshop — Tutorial D4

version **#v4.1.0**



**Yet Another Kind of Assistants 2022** <yaka@tickets.assistants.epita.fr>

# Copyright

This document is for internal use at EPITA (website) only.



# Contents



*https://intra.assistants.epita.fr

# 1 Serialization

## 1.1 What is it?

Serialization is a way to **save the current state of an object into a data stream** (generally a file). Deserialization is the opposite: **create an object with a state extracted from a data stream**.

You will need to serialize an object when, for instance, you wish to send it through a network.

In *Java*, there are different types of serialization:

- **binary** serialization;
- **JSON** serialization;
- **XML** serialization;
- **SQL** serialization;
- And many more ...

In this tutorial, you will be taught about *binary* and *JSON* serialization.

## 1.2 Binary serialization

First things first: for a class to be able to be serialized, it **must** implement the Serializable interface. This interface does not define any method: it only identifies the semantics of being serializable. All classes that inherit from a `Serializable` class are also `Serializable` themselves.

At runtime, a serializable class is associated to a `static final long` variable called **serialVersionUID** which is used to check that the classes loaded by both the sender and the receiver of a serialized object are compatible. If the receiver of a serialized object tries to deserialize it in an instance of a class with a `serialVersionUID` different from the object's, an `InvalidClassException` will be thrown.

```
public class Main {
    public static class Assistant implements Serializable {
        public static final long serialVersionUID = 47L;

        public final String name;
        public final String address;

        public Assistant(final String name, final String address) {
            this.name = name;
            this.address = address;
        }
    }
}
```

Though a default `serialVersionUID` is generated if none is defined, we **strongly advise** you to define one by yourself, as different *Java* compilers can generate a different `serialVersionUID` for the same class definition.

Serialization is done using a class named ObjectOutputStream, whose constructor takes an `OutputStream` as argument. You will mostly want to serialize objects into files, which can be done hassle-free: the `FileOutputStream` class inherits from `OutputStream`.

`ObjectOutputStream` provides methods to send data to the given `OutputStream`, such as `writeObject(Object o)`.

```
// Append this to your previous code-block!
public static void main(String[] args) {
    final var chef = new Assistant("Chef", "Laboratoire des assistants");

    // The standard extension for files created by binary serialization is '.ser'
    try (final var myOos = new ObjectOutputStream(new FileOutputStream("/tmp/chef.ser"))) {
        myOos.writeObject(chef);
    } catch (final IOException e) {
        System.err.println("Oops!");
        e.printStackTrace();
    }
}
```

If a field of a serializable class is not serializable (or if you don't want it to be serialized), you **must** declare it as `transient`. A `transient` attribute is ignored during serialization, and its value is set to null (or 0 for numerical types) during deserialization by default.

```
// Replace the previously defined Assistant class with this one!
public static class Assistant implements Serializable {
    public static final long serialVersionUID = 47L;

    public final String name;
    // Actually, I do not want people to know where assistants live...
    // Let us not serialize this one.
    public final transient String address;

    public Assistant(final String name, final String address) {
        this.name = name;
        this.address = address;
    }
}
```

Deserialization is the opposite operation: you will recreate your object from a file. It is done using the ObjectInputStream, whose constructor takes an `InputStream` as argument. `ObjectInputStream` provides the `readObject()` method to deserialize your file.

Beware, `readObject()` returns an `Object` class, therefore you must cast it in the type of the wanted object!

```
// Replace your previous main with this one!
public static void main(String[] args) {
    final var chef = new Assistant("Chef", "Laboratoire des assistants");
    System.out.println(chef.name + " lives at " + chef.address);

    // The standard extension for files created by binary serialization is '.ser'
    try (final var myOos = new ObjectOutputStream(new FileOutputStream("/tmp/chef.ser"))) {
        myOos.writeObject(chef);
    } catch (final IOException e) {
        System.err.println("Oops!");
        e.printStackTrace();
        return;
    }
```

(continues on next page)

(continued from previous page)

```
    try (final var myOis = new ObjectInputStream(new FileInputStream("/tmp/chef.ser"))){
        final var newChef = (Assistant) myOis.readObject(); // Notice the cast!
        // Address is transient, so set to null by default at deserialization.
        System.out.println(newChef.name + " lives at " + newChef.address);
    } catch (final IOException e) {
        System.err.println("Oops!");
        e.printStackTrace();
    } catch (final ClassNotFoundException e) {
        System.err.println("Might be an unmatching serialVersionUID...");
        e.printStackTrace();
    }
}

/*
 *  Output:
 *   Chef lives at Laboratoire des assistants
 *   Chef lives at null
 */
```

**Going further...**

If you have some special needs in the serialization of your class, you can override the `writeObject` method and define custom serialization behavior for your class. Similarly, you can override `readObject` if you want some specific behavior at deserialization. Redefining `readObject` is particularly useful when you want to compute the value of a transient attribute from other attributes.

```
// Replace your previous Assistant class with this one!
public static class Assistant implements Serializable {
    public static final long serialVersionUID = 47L;

    public final String name;
    private transient String address;

    public Assistant(final String name, final String address) {
        this.name = name;
        this.address = address;
    }

    public String getAddress() {
        return address;
    }

    private void readObject(ObjectInputStream inputStream) throws IOException,
                                                                  ClassNotFoundException {

        inputStream.defaultReadObject();
        this.address = "their home";
    }
}

/* Output (with the same main as previously):
 *  Chef lives at Laboratoire des assistants
```

(continues on next page)

(continued from previous page)

```
 *  Chef lives at their home
 */
```

## 1.3 JSON serialization

### 1.3.1 Definition

*JSON* stands for *JavaScript Object Notation*. It is a lightweight data interchange format. This text format is easy to read and write by humans, and is based on a part of the *JavaScript* programming language.

There are six different *JSON* types:

- Strings, in double quotes
- Numbers
- Booleans (*true* or *false*)
- *null*
- Objects which are unordered sets of name/value pairs. An object begins with "{" (left brace) and ends with "}" (right brace). Each name is followed by ":" (colon) and the name/value pairs are separated by "," (comma).
- Arrays which are ordered collections of values. An array begins with "[" (left bracket) and ends with "]" (right bracket). Values are separated by "," (comma).

Objects and arrays can be nested. This means that an array can contain other arrays or objects, and same goes with the objects.

**Going further...**

For more details about the *JSON* standard you may want to visit: json.org.

### 1.3.2 Jackson

Today you will get to know about Jackson: a library to process *JSON* in *Java*.

In order to add Jackson to your project's dependencies, add the following code inside the `<dependencies>` block of your `pom.xml` file:

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.10.0</version>
</dependency>
```

**Tips**

You can add the Jackson dependency in your `pom.xml` using the `Alt+insert` keyboard shortcut. Then select dependency, enter jackson and choose "**jackson-databind**". It will add the correct

dependency to your project with the latest version.

Using this class:

```
public class Person {
    public String firstName;
    public String lastName;
    public int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    // Jackson needs a default constructor to initialize the object
    public Person() {
    }
}
```

You would save its *JSON* representation to a `String` as follows:

```
final ObjectMapper mapper = new ObjectMapper();
final Person person = new Person("Jackson", "Me", 42);
final String jsonString = mapper.writeValueAsString(person);
```

How to retrieve an object from its *JSON* representation:

```
final ObjectMapper mapper = new ObjectMapper();
final String jsonString = "{\"firstName\" : \"Jackson\", \"lastName\" : \"Me\", \"age\" : 42}
↪";
final Person person = mapper.readValue(jsonString, Person.class);
```

**Going further...**

The previous examples only give you a glimpse of Jackson's features. To deepen your understanding, we strongly recommend you to have a look at the library's documentation.

## 1.4 Exercise

### 1.4.1 Serialize

#### Objectives

The goal of this exercise is to create a `Student` class, and to make it serializable in both *JSON* and binary formats.

Here is the base student class:

```
public class Student {
    private String firstname;
    private String lastname;
```

(continues on next page)

(continued from previous page)

```
    private String address;
    private int age;
}
```

## Specifications

You will first have to create the `Student` class, with a basic function that prints its identity as shown in the examples below.

Then, you will need to serialize and deserialize the `Student` using binary serialization. Those functions will be implemented in the class `BinarySerialization`.

You also have to implement an Exception called `SerializationException` that outputs an error message that begins with:

```
An error happened while serializing:
```

Finally, you have to implement the same operations, but using *JSON* serialization thanks to the Jackson library.

In both types of serialization, you should not serialize the address of the `Student`: it is a private information, that we don't want to share. Moreover, in order to verify that the deserialization worked correctly, a `Student` will always print his identity on the standard output when being deserialized.

## Prototypes

```
class Student {
    public Student(String firstName, String lastName, int age, String address);
    public void printIdentity();
}

class BinarySerialization {
    public static void serializeStudent(Student s, String file)
        throws SerializationException;
    public static void deserializeStudent(String file)
        throws SerializationException;
}

class JsonSerialization {
    public static String serializeStudent(Student s)
        throws SerializationException;
    public static void deserializeStudent(String json)
        throws SerializationException;
}
```

### Example

```
Student student = new Student("Bob", "Dupont", 21);
String fileOut = "student.tmp";
BinarySerialization.serializeStudent(bob, fileOut);
BinarySerialization.deserializeStudent(fileOut);
```

This should print (the symbol $ represents a newline character):

```
First name: Bob$
Last name: Dupont$
Age: 21$
Address: null$
```

### Warning

Your serialization and deserialization will be tested separately.

## 2 Logging

A **log** is an append-only, totally-ordered sequence of records ordered by time. Logs basically tell the story of the application that generates them. They are needed for multiple reasons:

- Cross-application standardization
- Flexibility

### 2.1 Implement logs in JAVA

We will use the Logging Framework `logback`[1] during this workshop. The following is the most simple example of how to produce logs with `logback` :

```
final Logger logger = LoggerFactory.getLogger(this.getClass());
logger.info("Hello World!");
```

Which will produce :

```
08:53:01.896 [main] INFO com.epita.scratch.Logging - Hello, world!
```

Notice the added information:

- Some form of timestamp
- The thread from which the log has been created
- The **logging level**
- The source logger (usually the name of the class)

[1] http://logback.qos.ch/setup.html

- The actual message

## 2.2 About log levels

Levels are used to specify the importance and granularity of the message. The logger can be configured to ignore messages lower than a given priority. By default, the logger's assigned log level is `DEBUG`.

Here is a list of all the usual log levels:

- `ALL`: lowest rank, everything will be printed
- `TRACE`: very fine grained tracking
- `DEBUG`: fine grained activity tracking for specific debugging purposes
- `INFO`: coarse grained activity tracking
- `WARN`: indicates that a suspicious situation has been encountered
- `ERROR`: indicates a recoverable error
- `FATAL`: indicates an error from which the program won't recover
- `OFF`: turns off logging entirely

## 2.3 Where to use logging

Logging should be used instead of any `System.out.println` and `System.err.println` you might already have in your project, for several purposes:

- In case of an anomaly, use FATAL, ERROR or WARN depending on the severity of the issue
- Anywhere a meaningful branch or decision is taken, use INFO
- For fine grained debugging, use DEBUG or TRACE

## 2.4 Logback basic formatting and features

`Logback` allows parameter expansion instead of string concatenation.

```
logger.warn("An error occurred while parsing string {} from user {}", string, user);
```

`Logback` handles exceptions properly. Most logging methods have an overload taking an exception as its last parameter.

```
catch(final Exception exception) {
  logger.error("Unknown error", exception);
}
```

### 2.5 To go further

Appenders are the components of `logback` responsible for outputting logging events. They can route messages to different targets:

- Files
- Network, using various protocols
- Databases
- Custom

Consider taking a look at the official `logback` documentation[2] if you encounter issues with any of the features described above or to discover others that go beyond the scope of what we covered in the workshop.

## 3 Asynchronous

To practice what you have seen during this morning class, let us dive into the `scheduler` exercise !

### 3.1 Objectives

You have to implement a scheduler, by implementing the MyTask class.

### 3.2 Examples

```
var bestShell = MyTask.of(() -> 42)
    .andThenWait(1L, TimeUnit.SECONDS)
    .andThenDo(value -> value + "sh")
    .execute();
    System.out.println(bestShell);
```

Will wait for a second and then print:

```
42sh
```

**Tips**

The `scheduler` given files can be found on the Assistants'intranet.

*Don't be afraid, I just wanna help you.*

[2] http://logback.qos.ch/manual/index.html