# Reading and Writing Data and Objects

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR
@JosePaumard https://github.com/JosePaumard

# Agenda

Concept of Serialization

Create binary images of Objects

How to override the standard mechanism

## Why Serializing Objects?

## About Serialization

Serialization is a general mechanism

It is about creating a portable representation of an object

That can be stored on a disk, for later use

Or sent over a network to another application

```java
public class Person {

    private String name;
    private int age;

    // some methods
}
```

This is a Java Person class, only Java code can use it

```java
public class Person {

   private String name;
   private int age;

   // some methods
}
```

```xml
<person>
   <name>Sarah</name>
   <age>32</age>
</person>
```

This is a Java Person class, only Java code can use it

This is an instance of Person, in XML, it is portable

```java
public class Person {

   private String name;
   private int age;

   // some methods
}
```

```json
{
   "Person": {
      "name": "Sarah",
      "age": 32
   }
}
```

This is a Java Person class, only Java code can use it

This is an instance of Person, in XML, it is portable

This is a JSON instance, also portable

XML and JSON are two ways of serializing Java objects in a portable way

But…

- XML is first published in 1998
- JSON is 2013 – 2014
- Java is 1995

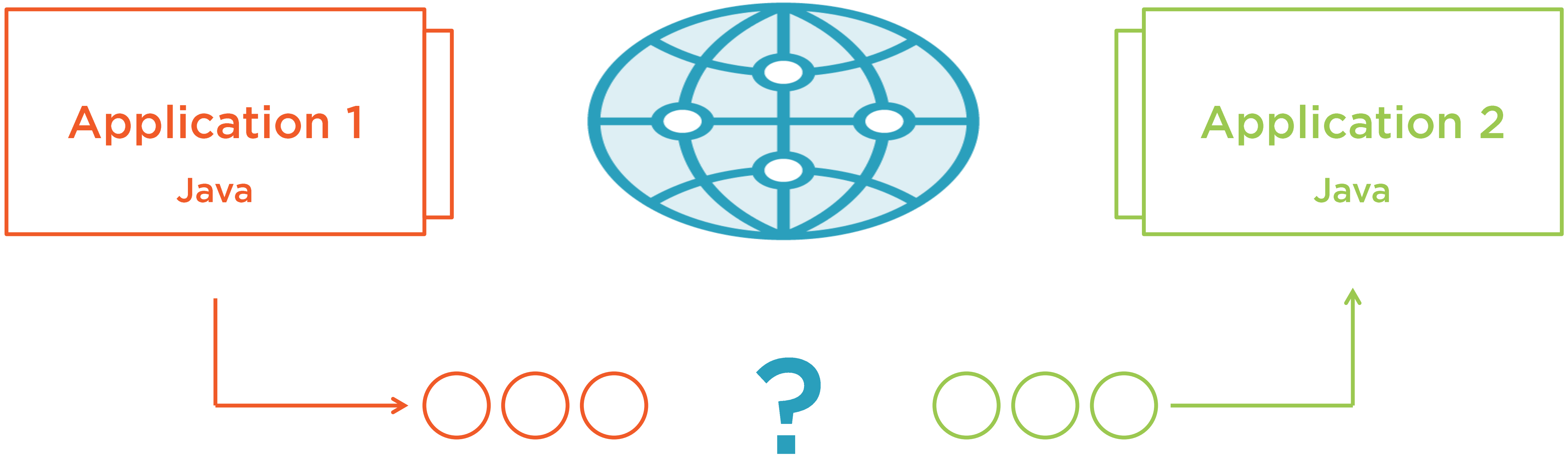
So Java has its own Serialization mechanism

Serialization in Java is vey widely used in the JDK and in Java EE

It is said to be very costly to maintain across the JDK versions

It may be quite complex to fully understand

But is very smooth to use



- The state
- The name of the class
- The version of the class

## Making a Class Serializable

Only instances of Serializable classes can be serialized

The only thing to do for the class is to implement the Serializable interface

Which has no method!

There is still one thing to do

Add a special static field: serialVersionUID

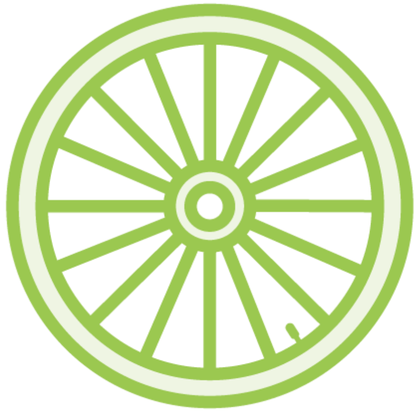
If it is not, then it will be computed when needed

```java
public class Person implements Serializable {

    private static final long serialVersionUID = 2027893533838449164L;

    private String name;
    private int age;

    // some methods
}
```

This is Person class that can be passed from one application to another

Across a network or through a disk

The serial version UID can be generated by all the IDE



How is the serial version UID computed?

The computation is fully specified in the Java Language Specification

It is a hash computed from the class name, interfaces implemented, methods and fields using a SHA

If the field is present in the class at compile time, then it will be used as is, with no validation

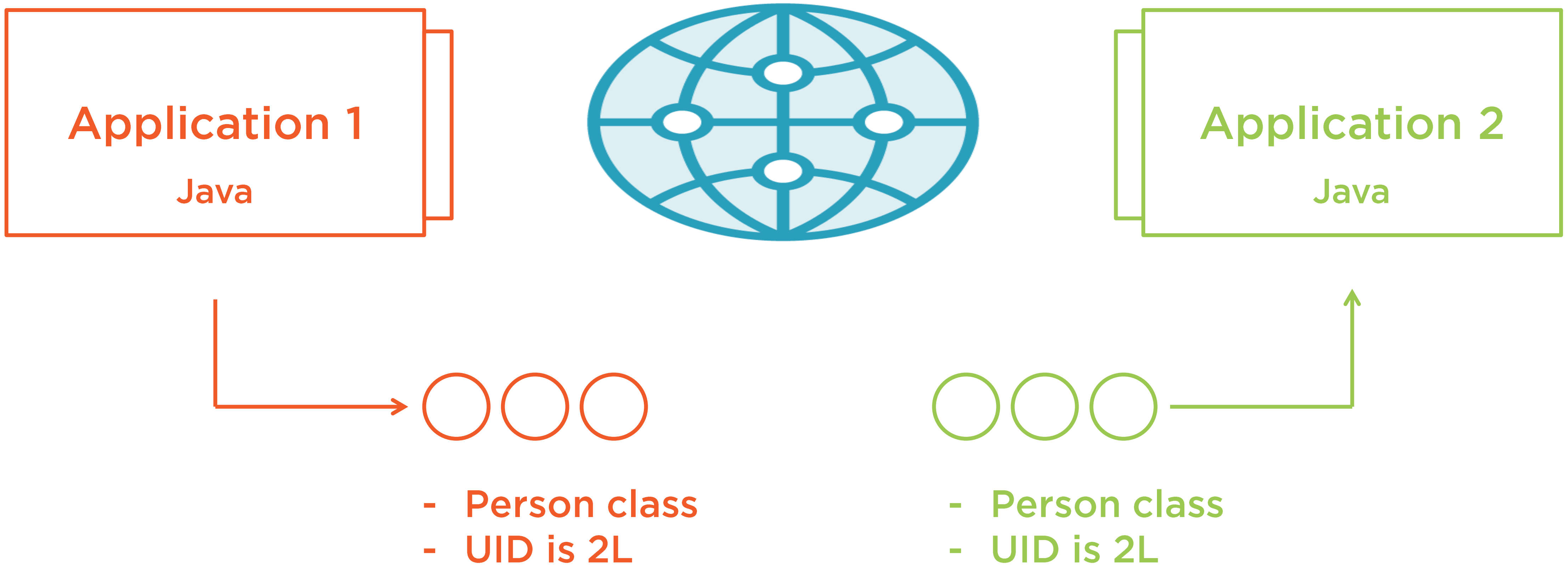
## Serializing and Deserializing Objects

Since we can force the value of the serialVersionUID field, 3 cases can occur:

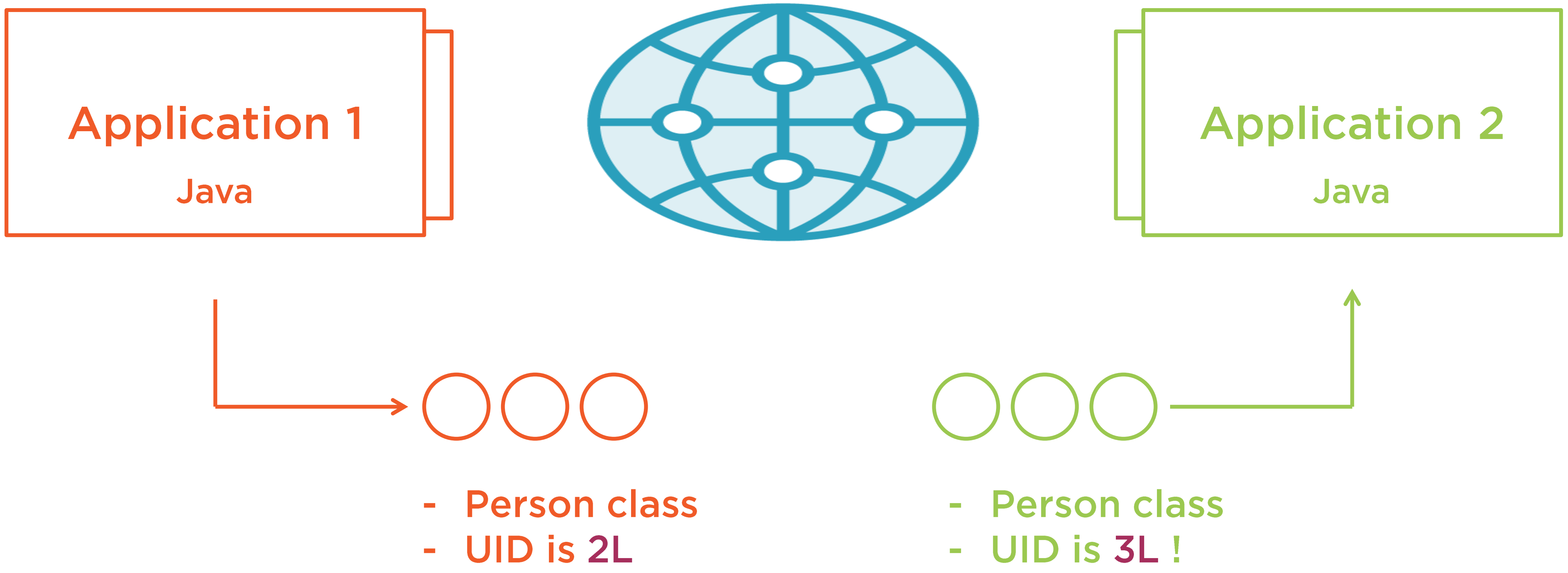
- the serialVersionUID is the same and the class is the same

- the serialVersionUID is the same and the class is not the same

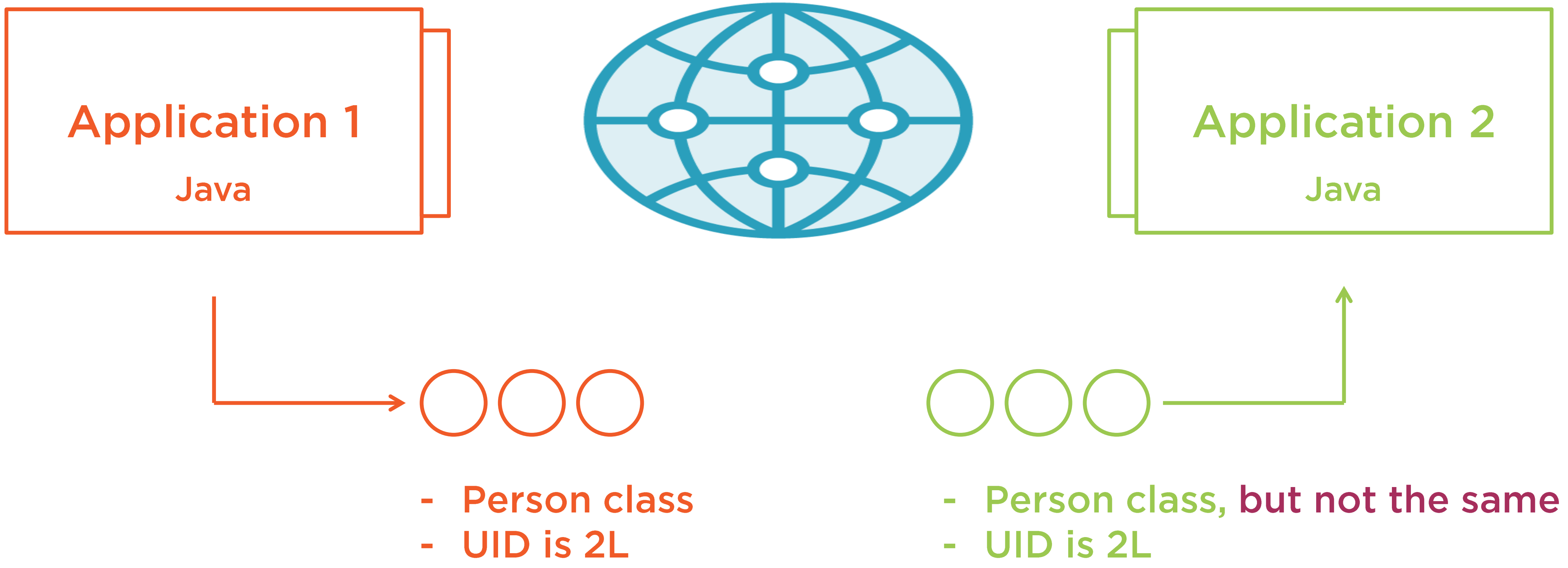
- the serialVersionUID is not the same



In this case, App 2 will deserialize the Person instances



In this case, App 2 will not deserialize the Person instances, raising an Exception



In this case, App 2 will deserialize
the Person instances, using default behavior

## Serialization / Deserialization

Can be used to exchange objects across a network

Is flexible enough to deserialize objects in almost the same class, in a controlled way

## Writing Serialization Code

```java
OutputStream os = Files.newOutputStream(
   Paths.get("files/people.bin"),
   StandardOpenOptions.CREATE
);


ObjectOutputStream oos = new ObjectOutputStream(os);

oos.writeObject(person1);
oos.writeObject(person2);
```

Writing a serialized object to an output stream is so simple!

```java
InputStream is = Files.newInputStream(
   Paths.get("files/people.bin"),
   StandardOpenOptions.READ
);


ObjectInputStream ois = new ObjectInputStream(is);

Person person1 = (Person)ois.readObject();
Person person2 = (Person)ois.readObject();
```

And reading is the same!

```java
public class Person implements Serializable {

    private String name;
    private int age;

    private Address address;
}
```

By default, relations are serialized too…

So if Address is not serializable, an Exception will be raised

```java
public class Person implements Serializable {

    private String name;
    private int age;

    private transient Address address;
}
```

By default, relations are serialized too…

So if Address is not serializable, an Exception will be raised

To prevent that, we can use the transient keyword

The default serialization mechanism

- stores all the fields of the object

- apart from the transient ones

- if a field is neither transient nor serializable, an exception is raised

## Overriding Default Serialization

There 3 ways to override the default serialization mechanism

- providing a pair of writeObject() / readObject() methods

- implementing the Externalizable interface

- providing the writeReplace() and readResolve() methods

```java
public class Person implements Serializable {

    private String name;
    private int age;

    private void writeObject(ObjectOutputStream oos) throws Exception {
        // some code
    }
}
```

1st solution: readObject / writeObject

The writeObject method must exactly match this one

Its responsibility is to write the fields of this Person class on the provided object stream

```java
public class Employee extends Person {

    private int salary;


    private void writeObject(ObjectOutputStream oos) throws Exception {
        // some code
    }
}
```

In case Employee extends Person (then Employee is Serializable)

The Employee.writeObject method will also be called

The role of writeObject is to handle its class, not the super classes

```java
public class Person implements Serializable {

    private String name;
    private int age;

    private void readObject(ObjectInputStream ois) throws Exception {
        // some code
    }
}
```

1st solution: readObject / writeObject

And the same goes for readObject

Its responsibility is to read the fields of this Person class on the provided object stream

```java
public class Employee extends Person {

   private int salary;


   private void readObject(ObjectInputStream ois) throws Exception {
      // some code
   }
}
```

In case Employee extends Person, the Employee.readObject method will also be called

The role of readObject is to handle its class, not the super classes

Of course readObject must read what writeObject wrote

1st solution: writeObject / readObject

- the methods must exactly match the specified methods

- they handle the class they are in, not the super classes

- they must be compatible with each other

```
public interface Externalizable {

    void writeExternal(ObjectOutput out)
    throws IOException;

    void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException;
}
```

2nd solution: implement Externalizable

A class that implements Externalizable is serializable

The methods used to serialize the class are public

They must serialize the class and its super classes (if any)

2nd solution: implement Externalizable

- provide a pair of read / write public methods

- made to serialize only the identity (primary key) of an object, not its state

- the receiving side can then recreate the state of the object from its class and primary key

```java
public class Person implements Serializable {

    private String name;
    private int age;

    private Object writeReplace() throws ObjectStreamException {
        return new PersonProxy(name + "::" + age);
    }
}
```

## 3rd solution: use proxy objects

The class has a writeReplace method (private, protected or public)

This method returns the object that will be serialized

```java
public class PersonProxy implements Serializable {

    private String replacer;


    public PersonProxy(String replacer) {
        this.replacer = replacer;
    }
}
```

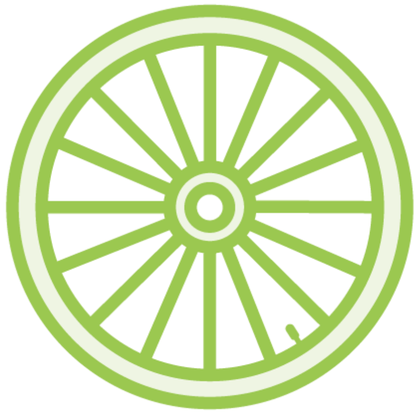
Of course the proxy class must be serializable

It can use an overridden serialization

```
public class PersonProxy implements Serializable {

    private String replacer;


    private Object readResolve() throws ObjectStreamException {
        String[] elements = replacer.split("::");
        String name = elements[0];
        int age = Integer.parseInt(elements[1]);
        return new Person(name, age);
    }
}
```

And the proxy class must provide a readResolve method, that can create the real object from the proxy

The serialization will deserialize the proxy object, then call this readResolve method and return the result



3rd solution: use a proxy object

- provide a pair of writeReplace / readResolve methods

- the proxy object is stored in the serialized stream

- transparent for the caller

- can also be used handle different versions of a serializable class

There 3 ways to override the default serialization mechanism

- providing a pair of writeObject() / readObject() methods

- implementing the Externalizable interface

- providing the writeReplace() and readResolve() methods

## Overriding Serialization

Three ways to do that

Can handle all the use cases we can find in applications

## Demo

Let us see some code!

Let us play with readObject / writeObject

And see how we can use proxies to serialize and deserialize objects

# Module Wrap Up

What did you learn?

Object Streams!

How to write and read objects

How to override standard serialization to finely control the object streams