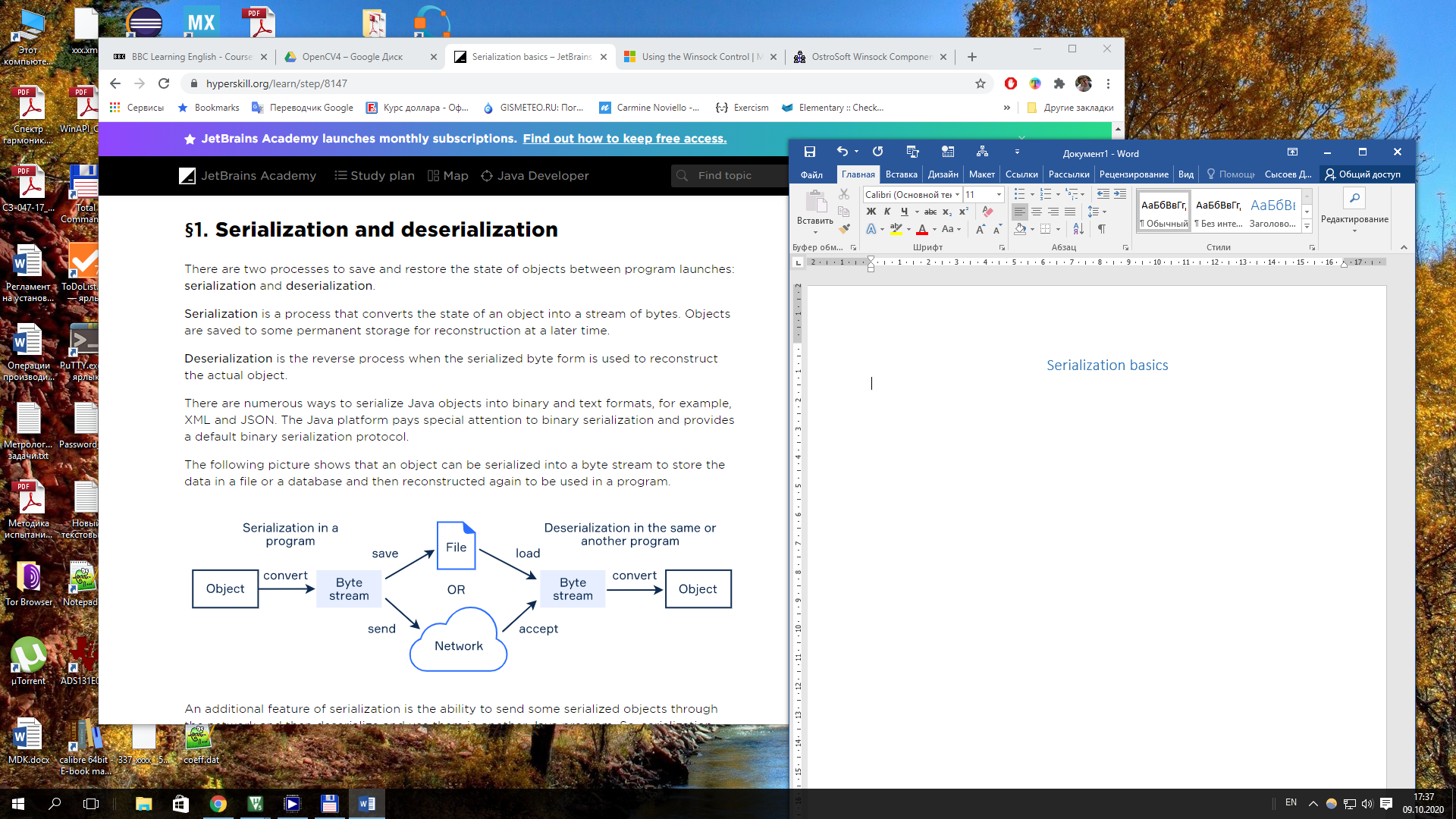
# Serialization basics



# 2. Making a class serializable

To make a class serializable, it must implement the Serializable interface. This is a **marker interface** without methods. This is used to inform the compiler that the class implementing it has some special behavior.

class SomeClass implements Serializable {

}

The class being serialized can contain any primitive type and any other class as its field. All related values and objects will also be serialized. You can easily **prevent** a field from being serialized with the transient keyword:

private transient String nonSerialziedField;

There is a recommendation to add a special field called serialVersionUID for every class that implements this interface. The field should be static, final and long.

private static final long serialVersionUID = 7L;

The serialVersionUID field is used to verify that the sender and the receiver of a serialized object are compatible and have loaded the classes for that object. If the version number of the sender and receiver classes don’t match, then the runtime error InvalidClassException occurs. The matching of this value happens “under the hood” during serialization and deserialization. An error occurs only in cases of mismatch.

Although it's not required, it is strongly recommended for a serializable class to explicitly declare its own serialVersionUID. Declaring and using this number guarantees a consistent serialVersionUID value across different Java compiler implementations. At the same time, there is no need for two different classes to have unique values for this field.

# 3. Streams for objects

In java, the serialization and deserialization mechanisms are based on the standard I/O system and byte streams. They use the ObjectOutputStream and ObjectInputStream classes accordingly.

The first class provide a method called void writeObject(Object object) which writes the state of the specified object to the stream. The second class has a corresponding method Object readObject() to restore the object. Both methods throw exceptions when something is wrong.

Example:

class SerializationUtils {  
    /\*\*  
     \* Serialize the given object to the file  
     \*/  
    public static void serialize(Object obj, String fileName) throws IOException {  
        FileOutputStream fos = new FileOutputStream(fileName);  
        BufferedOutputStream bos = new BufferedOutputStream(fos);  
        ObjectOutputStream oos = new ObjectOutputStream(bos);  
        oos.writeObject(obj);  
        oos.close();  
    }  
  
    /\*\*  
     \* Deserialize to an object from the file  
     \*/  
    public static Object deserialize(String fileName) throws IOException, ClassNotFoundException {  
        FileInputStream fis = new FileInputStream(fileName);  
        BufferedInputStream bis = new BufferedInputStream(fis);  
        ObjectInputStream ois = new ObjectInputStream(bis);  
        Object obj = ois.readObject();  
        ois.close();  
        return obj;  
    }  
}

# An example: citizens

Suppose, you need to develop an information system that persistently stores all citizens of a country. Here are two related classes for this system. Citizen and Address. Both classes implements the Serializable interface and contain the serialVersionUID fields

public class Citizen implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    private String name;  
    private Address address;  
    private transient String passport;  
  
    // getters and setters  
  
    @Override  
    public String toString() {  
        return "Citizen{" +  
                "name='" + name + '\'' +  
                ", passport='" + passport + '\'' +  
                ", address=" + address +  
                '}';  
    }  
}

class Address implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    private String state;  
    private String city;  
    private String street;  
  
    // getters and setters  
  
    @Override  
    public String toString() {  
        return "Address{" +  
                "state='" + state + '\'' +  
                ", city='" + city + '\'' +  
                ", street='" + street + '\'' +  
                '}';  
    }  
}

The value of serialVersionUID of a class should be increased whenever you make a change that adds/updates/removes a field. Otherwise, you may encounter exceptions during the deserialization process for objects which were saved before this change.

# Serializing and deserializing objects

Here is a method that generates an array of citizens.

public static Citizen[] initCitizens() {  
    Citizen mark = new Citizen();  
    mark.setName("Mark Olson");  
    mark.setPassport("503143798"); // the passport was set  
  
    Address markAddress = new Address();  
    markAddress.setState("Arkansas");  
    markAddress.setCity("Conway");  
    markAddress.setStreet("1661  Dawson Drive");  
  
    mark.setAddress(markAddress);  
  
    Citizen anna = new Citizen();  
    anna.setName("Anna Flores");  
    anna.setPassport("605143321"); // the passport was set  
  
    Address annaAddress = new Address();  
    annaAddress.setState("Georgia");  
    annaAddress.setCity("Atlanta");  
    annaAddress.setStreet("4353  Flint Street");  
  
    anna.setAddress(annaAddress);  
  
    return new Citizen[]{ mark, anna };  
}

Here is the main method which runs all the work and uses the SerializationUtils class.

public static void main(String[] args) {  
    String filename = "citizens.data";  
    try {  
        SerializationUtils.serialize(initCitizens(), filename);  
        Citizen[] citizens = (Citizen[]) SerializationUtils.deserialize(filename);  
        System.out.println(Arrays.toString(citizens));  
    } catch (IOException | ClassNotFoundException e) {  
        e.printStackTrace();  
    }  
}