Serialization Instuctions

What is serialization?

Serialization is the process of converting objects into a more generic data format that can be stored or sent across the internet.

When your program is running, your Java objects are available only locally on the Java Virtual Machine that is running your program. Once the program ends, that data is gone. If you want to send that data to another program or persist them on your computer, you need to serialize the data.

In the following example, I will show you how to serialize data into a byte stream. Another popular serialization method is JSON. If I have time later and you're interested, I can make a tutorial about that as well 😃

Step 1: Implement serializable

The first step to serializing is implementing the serializable interface on any classes you want to serialize. This is as simple as adding the `implements` keyword and `Serializable` interface as part of the class declaration.

```
public class Bike implements Serializable {
    private double tireWidth;
    private boolean hasSuspension;

    public Bike(double tireWidth, boolean hasSuspension) {
        this.tireWidth = tireWidth;
        this.hasSuspension = hasSuspension;
    }
}
```

Step 2: Serialize object(s)

To serialize an object of the `Bike` class in the above example, we will use `FileOutputStream` and `ObjectOutputStream`.

`FileOutputStream` takes a path to the file. If the file doesn't exist, it will be created. To create the file in the root of your project, just type the filename. The `.ser` extension is used to indicate a serialized file.

```
FileOutputStream fileOutputStream = new FileOutputStream(bikes.ser);
```

`ObjectOutputStream` will actually do the writing. It takes the newly created `FileOutputStream` to instantiate. Once the object is instantiated, you can use it's methods to write your file.

```
ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);

// This is where you pass your object, or your List of objects
objectOutputStream.writeObject(bikeObject);

// Close out the stream
objectOutputStream.close();
```

You will notice IntelliJ prompts you to wrap all this in a `try/catch`. `FileOutputStream` can throw a `FileNotFoundException` and `ObjectOutputStream` can throw an `IOException`.

Note that if you go to view your `.ser` file it will be full of nonsense. This is because the data is being stored as bytes, and not in a human readable format.

Step 3: Deserialize object(s)

Now that you have a serialized file you can write code that will read from the `.ser` file and dersialize that data back into normal Java objects.

It's a very similar format to serializing. Instead of using classes with output in the name, they will have input in the name.

```
FileInputStream fileInputStream = new FileInputStream(bikes.ser);
ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);

// The (Bike) is typecasting what comes from the input stream into a Bike
Bike bike = (Bike) objectInputStream.readObject();
```

If you are serializing/deserializing a list, you may need to do something like this:

```
// This is typecasting, but is an unchecked cast. It will work, but may have a
warning in IntelliJ
ArrayList<Bike> bikeArrayList = (ArrayList<Bike>) objectInputStream.readObject();
```

Just like with our output, this also will need to be surrounded in a `try/catch` statement, for the same exceptions as before.

Now you try!

Using the above examples, write functions that can serialize and deserialize objects.