Lab 3: Object Construction, Serialization and Deserialization, Access Control

# Problem 1: Object Construction

## Part 1: Create a static factory method (“static constructor”)

You are being given two files: **User.java** and **UseUser.java**. The **User.java** file contains the definition of a user consisting of two attributes (**username** and **password**), setter and getter methods for the attributes, and a **validate** method that can be used to check provided user details against the attribute values.

The **UseUser.java** file contains a **main** method that instantiates three **User** objects and prints out their attribute values.

The User class uses normal constructor methods to instantiate User-type objects. Examine the code given to you, compile and run it. Examine the output produced by the **UseUser** class.

Create a new class called **UserStaticConstructor**. This class should work the same as the **User** class, but it should provide *public static* factory methods called **getInstance()** to instantiate and return an object of type **UserStaticConstructor**. If a provided username or password value is ***null*** in the factory methods then the method should throw an **IllegalArgumentException** with a message of “***Missing user data***”.

Once you have implemented the **UserStaticConstructor** class create a file called **UseUserStaticConstrutor.java**. This file contains a **main** method that attempts to instantiate three **UserStaticConstuctor** objects, including one that is given a ***null*** username and/or password. Output the data in each of the instantiated objects.

**Question: What is the difference between using User versus UserStaticConstructor objects with respect to object creation?**

## Part 2: “Fix” the User class

Create a new version of the **User** class from problem 1 that includes an “***initialized***” flag attribute. The initialized flag should be set to **true** only if all of the attributes of an instance of a **User** are initialized properly during construction. If this does not happen then the getter methods of the class should always return **null** values.

Write a **main** method to instantiate and test your new **User** class definition. You can deliberately throw errors in the **User** class constructors to force an instance of **User** to be improperly initialized.

# Problem 2: Serialization and Deserialization

## Part 1: Serialize and Deserialize an object

You are being given a files **User.java, SerializeUser.java, and DeserializeUser.java.**  User.java is the same as the **User** class from problem 1, but it also implements the **Serializable** interface.

Run SerializeUser and DeserializeUser. Examine the contents of the **user.ser** file and interfere with the values of the attributes saved into it so that when you deserialize it the values are changed in the deserialized object in memory.

## Part 2: *Transient* attributes in serialized objects

Create a new version of the **User** class called **UserTransient**. This new class makes the password attribute a *transient* field with respect to serialization and deserialization.

Create a new version of your **SerializeUser** class called **SerializeUserTransient**. This class should instantiate a **UserTransient** object and save a serialized copy of it into a file called **usertransient.ser**.

Create a new version of your **DeserializeUser** class called **DeserializeUserTransient**. Use it to deserialize the contents of the **usertransient.ser** file and show the values of the attributes of the deserialized object.

## Part 3: Use an ObjectStreamField array (“serialPersistentFields”) to mimic *transient*

Create a copy of the **UserTransient** class called **UserSerialPersistentFields**. Create in it another attribute of type **ObjectStreamField[]** which specifies that only the **username** attribute should be serialized. Remove any reference to “***transient***” from this class.

Create copies of your **SerializeUserTransient** and **DeserializeUserTransient** classes called **SerializeUserSerialPersistFields** and **DeserialUserSerialPersistFields**. The first class should serialize a **UserSerialPersistentFields** object into a file called **userserialpersistfields.ser**. The second class should deserialize that file and print out the attribute values of the deserialized object. Make sure that only the **username** attribute is being serialized and deserialized.

# Problem 3: Intersection of Permissions during Access Control

## Part 1

The Problem 3/Part 1 directory contains the following files:

* A text file called **input.txt** (which the program will read in and print out the contents on-screen)
* A directory called **main** containing a class called **TestMe**. It also contains a file called **commandline.txt** which shows you how to compile and run the application (from this directory), and a file called **sec.pol** which contains Java security manager permissions to be used when running the application.
* A directory called **other** containing a .jar file called **ReadFile.jar** which is created using the contents of the subdirectory of **other** called **MyPackage**.

The **ReadFile.jar** file contains a class called **TestClass()** which is part of **MyPackage**. This class contains a method called **test()** which attempts to opens and read the **input.txt** file and print out its contents on-screen (permissions allowing).

The **sec.pol** file contains the following permission settings:

grant codeBase "file:./\*" {

permission java.io.FilePermission "../input.txt", "read,write";

};

grant codeBase "file:../other/\*" {

permission java.io.FilePermission "../input.txt", "read";

};

The “**codeBase**” setting in the **grant** commands specifies the location that the permissions apply to. So in the above any classes loaded from “file:./\*” are allowed to read and write the **input.txt** file, but classes loaded from “file:../other/\*” can only read from it.

The *intersection of the permissions* as given to you in **sec.pol** allow the code in **MyPackage** to read from the **input.txt file**, but not to write to it. Code in **main** can write to the file. Confirm that is the case by editing the **main** method in **TestMe** to write new data to the file (which should work), and then by attempting the same in the **test()** method in **TestClass** from **MyPackage** (this should fail).

Once you have confirmed that the permissions are working as expected, edit the **sec.pol** file to first allow any operations that did ***not*** work initially, and then the disallow operations that ***did*** work initially. Run the program after each change to see what effect your changes have. This gives you practice in editing and applying security policy files in Java.

(Since this is practice only, just reply “done” to the discussions for this part.)

## Part 2: Using *doPrivileged*

This directory contains basically the same code as in Part 1, with one difference:

* In the **sec.pol** file the permissions for code loading from codebase “***file:./\****” have been commented out, meaning code from that location cannot access the **input.txt** file. ***Do not change the contents of this file for this part of the problem!***

To allow the code from “***file:.. /other/\****”to open and read the **input.txt** file you will need to use ***AccessController.doPrivileged*** to override the normal intersection of permissions used by Java so that it is acceptable for code from **TestClass** in **MyPackage** to open and read the file.

Edit the code in the **test** method in **TestClass** from **MyPackage** so that it uses **doPrivileged** to allow the permissions from **sec.pol** related only to “***file:../other/\****” to apply, and thus allow the program to read the contents of the file when normally it would not be allowed to do so (because of Java’s normal “*intersection of permissions*” access rule).