

COP2805C Module 8 Practice Exercise

In this practice exercise we will use Java I/O and NIO.2 APIs.

We repeatedly generate our user data each time we run the application, let's make that data persistent so we only generate it once. Our first choice would be to save it to a database, but we'll leave that for the upcoming JDBC module. For this module we will take advantage of our new-found knowledge about the I/O APIs by saving it to a file. There are lots of options here, we could write the information out as text in the form of a comma-separated-values file (CSV), as JSON, as XML, or some other textual format. Instead, let's take advantage of the serialization properties of Java objects by writing a binary form of the user information -- the objects themselves. We will use the legacy I/O classes [ObjectOutputStream](#) to serialize/write our data and [ObjectInputStream](#) to deserialize/read our data. We will store this data in a binary file, so I am using an extension of ".dat" for the file, a general-purpose indicator of non-textual data for the Windows platform (i.e. don't open this file with Notepad). In the HappyBirthday application class I have added a method to read an ArrayList of users from the data file and a method to write the ArrayList of users to the data file. Here is the reader:

```
// read saved user ArrayList
public ArrayList<User> readUsers() {
    ArrayList<User> list = new ArrayList();

    try (ObjectInputStream userData =
        new ObjectInputStream(
            new FileInputStream(USER_FILE))); {
        list = (ArrayList<User>) (userData.readObject());
        for (User u : list)
            System.out.println("readUsers: read " + u);
    } catch (FileNotFoundException e) {
        // not a problem if nothing was saved
        System.err.println("readUsers: no input file");
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return list;
}
```

We start off with a try-with-resources, this will ensure the data file is closed when we finish (we can do this because the ObjectInputStream class implements the [AutoCloseable](#) interface). The userData variable is instantiated to an ObjectInputStream which wraps a FileInputStream object (USER_FILE is just a String constant containing our file name). Get used to this wrapping syntax, it is very commonly used in real-world code along with chaining (e.g. object.method().method().method()...)

From here several things can happen.

1. The file exists and we read the contents normally. We know what the file is supposed to contain -- an ArrayList of User objects, so we have to cast the stream's readObject method to

what we expect it to be (otherwise it just returns an Object). If it doesn't turn out to be what we expect, an **invalidClassException** is thrown (and caught by our IOException handler).

2. The file does not exist and a **FileNotFoundException** is thrown. This is not a show-stopper, if it is the first time the application is being run it is expected, we will generate the data and create the file when we exit the application.

3. A **ClassNotFoundException** is thrown if the deserialized object is not Serializable. ArrayList implements Serializable, but we need to specify the User class is also serializable since that is the data type of our ArrayList contents.

```
public class User implements Serializable {
```

Finally, I am printing the users that are read as diagnostic output, I would probably get rid of this for the production version of the code.

In the main method for the HappyBirthday application class I have added logic which will generate the users if the file was not found or no users were stored in it:

```
// if no users, generate some for testing  
if (userList.isEmpty()) {
```

Here is the writeUsers method. Once again we wrap a File stream with an Object stream, but this time we are using output streams. And again, we use a try-with-resources so we do not have to manually close stream when we're done writing to it. The writeObject method does not require any casts, we write the entire ArrayList object in one simple operation.

```
// write user ArrayList to save file  
public void writeUsers(ArrayList<User> ul) {  
    try (ObjectOutputStream userData = new ObjectOutputStream(  
        new FileOutputStream(USER_FILE));) {  
        userData.writeObject(ul);  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

We write the data in the main method just before exiting (the created users are now saved in an ArrayList instead of separate local variables).

Now let's move to the second part of this exercise. A critical component of virtually all non-trivial applications is the ability to log events to a file for application monitoring and maintenance. This is part of what is known as **instrumentation**, how we measure software's behavior in order to diagnose errors and improve performance.

There are several publicly available logging packages for Java, e.g. the java.util.logging package and Apache Log4j (oops -- Log4j was recently the source of a critical security flaw), but we will create our own logging mechanism here to practice our file operations.

We will use a the NIO.2 [Files](#) class to create a [BufferedWriter](#) to write our log entries. Files is an NIO.2 class, but BufferedWriter is a legacy I/O class. We use BufferedWriter as a more efficient way to write text.

Some items to note here: Instrumentation can result in a performance hit so we need to very selective about what we log. Also, if we have multiple threads logging to the same file we may also need to consider synchronizing or otherwise protecting the writer, or maybe even creating a separate application devoted to logging, e.g. a microservice.

I created a generic interface for objects which will log information:

```
package edu.fscj.cop2805c.log;

public interface Logger<T> {
    public void log(T logObj);
}
```

Usually the log entry will simply consist of a String but this is more flexible to allow a specific implementation to extract information from any object.

So what should we log? There are two main events that would be interesting: when we create a user, and when we send a birthday greeting. For now we will keep it simple and just log the birthday greetings, so I have specified that the BirthdayCardProcessor class will implement the Logger interface:

```
public class BirthdayCardProcessor extends Thread
    implements MessageProcessor, Logger<BirthdayCard> {
```

Here's the Logger.log method implementation:

```
@Override
public void log(BirthdayCard c) {
    LocalDateTime local = LocalDateTime.from(
        Instant.now().atZone(ZoneId.systemDefault()));
    String msg = local.truncatedTo(ChronoUnit.MILLIS) +
        ":greeting:" + c.getUser().getName();
    try (BufferedWriter cardlog =
        Files.newBufferedWriter(Path.of(LOGFILE),
            StandardOpenOption.APPEND);) {
        cardlog.write(msg);
        cardlog.newLine();
    } catch (IOException e) {
        System.err.println("LOG FAIL" + msg);
        e.printStackTrace();
    }
}
```

We want a timestamp for any log entries, this is very important for when we need to diagnose problems. The timestamp created uses the local timezone, but sometimes it might need use UTC/GMT time so it is independent of the location. We will put a simple "greeting" prefix for the operation and

concatenate the user's name. We then open the logfile in append mode using the [newBufferedWriter](#) method of the Files class.

Why are we calling `newline()` instead of just using `"\n"` in the String? That is a way to use the current platform's line separator property which may or may not be `"\n"`. Not really necessary for us to use it here, but just to demonstrate how it can be used.

The log method is called from the `processMessage` method:

```
// remove messages from the queue and process them
public void processMessages() {
    System.out.println("before processing, queue size is " +
        safeQueue.size());
    safeQueue.stream().forEach(e -> {
        // Do something with each element
        e = safeQueue.remove();
        System.out.print(e);
        log(e);
    });
    System.out.println("after processing, queue size is now " +
        safeQueue.size());
}
```

Here is the output for the application:

```
readUsers: read User: Dianne Romero, Birthday: 2023-03-
18T20:45:43.039838900-04:00[America/New_York]
readUsers: read User: Sally Ride, Birthday: 2023-03-
19T20:45:43.039838900-04:00[America/New_York]
readUsers: read User: René Descartes, Birthday: 2023-03-
19T20:45:43.039838900-04:00[America/New_York]
readUsers: read User: Johannes Brahms, Birthday: 2023-03-
19T20:45:43.039838900-04:00[America/New_York]
readUsers: read User: Charles Kao, Birthday: 2023-03-
19T20:45:43.039838900-04:00[America/New_York]
before processing, queue size is 0
after processing, queue size is now 0
Users were read from data file
Dianne Romero
Sorry, today is not their birthday.
Sally Ride
René Descartes
Johannes Brahms
Charles Kao
polling
before processing, queue size is 4
Birthday card for Sally Ride
|-----|
```

```

|           Mar 19, 2023           |
|   Happy Birthday Sally Ride   |
| Hope all of your birthday wishes come true! |
|-----|
| Birthday card for René Descartes |
|-----|
|           19 mars 2023           |
|   Joyeux Anniversaire René Descartes |
| Hope all of your birthday wishes come true! |
|-----|
| Birthday card for Johannes Brahms |
|-----|
|           19.03.2023           |
|   Alles Gute zum Geburtstag Johannes Brahms |
| Hope all of your birthday wishes come true! |
|-----|
| Birthday card for Charles Kao |
|-----|
|           2023年3月19日           |
|           生日快乐 Charles Kao           |
| Hope all of your birthday wishes come true! |
|-----|
after processing, queue size is now 0
polling
before processing, queue size is 0
after processing, queue size is now 0
poll thread received exit signal

```

Here are the entries from the log file. Note that this log will continuously grow, log file management is an important requirement that we are not taking responsibility for in this application, but in a production environment we would need to write code to manage the log file by backing it up, compressing it, and more.

```

2023-03-19T21:18:45.374:greeting:Sally Ride
2023-03-19T21:18:45.378:greeting:René Descartes
2023-03-19T21:18:45.379:greeting:Johannes Brahms
2023-03-19T21:18:45.380:greeting:Charles Kao

```