**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

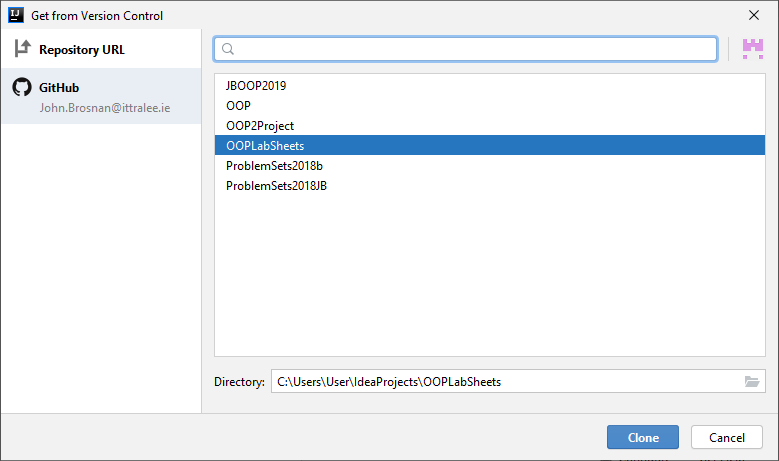
**Practical 15 – Files & Streams**

In this lab sheet, you will be introduced to the important concept of **persistence**, that is, making the data created/used within a program last or “persist” beyond the lifetime of that program. In Java, persistence is made possible through the use of **files** and **streams**.

**Getting into IntelliJ**

Launch IntelliJ. As you were introduced to **VCS** already, and since you should really have the latest version of your **OOPLabSheets** project “pushed” to GitHub, click “**Get from Version Control**” and see if you can now clone your **OOPLabSheets** project locally (if you haven’t your latest version pushed to GitHub, just copy it from your your X: drive to some location on C: or work directly with it from the memory stick).

IntelliJ will remember previously “pushed” repositories, so you can just pick off the one you want. You can also decide where you want the repository to be located locally, I am choosing the same location as the original,



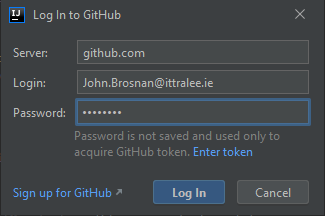
Recall that you **may need to rename** the local repository to something else if you get an error message at this point (I suggest a date e.g. OOPLabSheets27-11-20).

**Pushing an IntelliJ Project to GitHub**

**At the end of the lab session** you should really “**push**” your version-controlled OOPLabSheets project to GitHub, so that it then becomes a remote repository and a back-up of your work. I suggest you back it up to X: or memory stick also.

To do this, within IntelliJ just select **VCS**, then **Git**, then **Push** or, even better, just press the Git “Push” green arrow at the top of the IntelliJ window.

At this point, a dialog may pop up requesting your GitHub account details (IntelliJ might remember these also though). Once supplied, you can press the **Log In** button



If everything goes to plan, you will get a “Pushed 1 commit to origin master” message at the bottom-right of the IntelliJ window:

Now, for proof that the project is actually on GitHub, you can just **view your GitHub repositories list** and you should see the **OOPLabSheets** project listed. You can click into this then to make sure the latest files are definitely there.

**Setting up your Folder Structure**

My own preferred approach is to create a new folder for each lab sheet. In IntelliJ this can be done by adding a new **package** to the project. The package will be given an appropriate name, I will call it **labsheet15** here. Recall that **a Java package is simply a way to store related classes together** and essentially a **package is just a folder**. We will talk about packages further in this module but, for now, we will just create a package called **labsheet15** for this IntelliJ project and our intention will be to store all the related classes that we create and use for this lab sheet together within that package (folder). Right-click on the name of the project i.e. **OOPLabSheets** and select **New**🡪**Package.**

You will now be given the opportunity to enter the name of the package, so you can enter **labsheet15**. As soon as you click **OK**, an icon for the newly created package appears in the left-side window, listed as part of the project’s contents. The package is currently empty, but you will be adding some sub-packages to it as the lab goes on.

**Files**

As you know well, we use files every day on our computer to store information. We are able to amend the contents of files, remove files, create files, move files etc. We take it for granted that, when we modify a file and save those changes, the changes will be reflected the next time we go to open that file. This very document is an example of a file. All files are stored on some kind of medium, such as a hard-drive, a memory stick, DVD etc and so, when we need to process a file, we need to be able to access it successfully from one of these media.

We are going to examine how files are created, updated and processed by Java programs. Naturally enough, **file processing is one of the most important capabilities** for any programming language, so that it can support commercial applications that typically process massive amounts of “persistent” data.

**Streams**

Java views a file as a **sequential stream** of 8-bit bytes. When a file is opened, an object is created and a **stream** is associated with the object. Two stream objects that you are familiar with, and are **automatically provided** by the Java system, are **System.out** and **System.in**. As you know System.in is the **standard input stream** object, which enables a program to receive keyboard input, while System.out is the **standard output stream** object, allowing us to display information to the screen.

Java uses just two kinds of stream – **input streams**, which have a **source** from which the data comes and **output streams**, which have a **destination** to which the data goes.

To perform file processing in Java, the package **java.io** must be imported. This contains the **file stream classes** such as **FileInputStream**, for getting input from a file, and **FileOutputStream**, for sending output to a file. To perform input and output of objects and data types, the classes **ObjectInputStream**, **ObjectOutputStream**, **DataInputStream** and **DataOutputStream** are used together with the file stream classes.

There are many different kinds of streams defined within the Java API as seen in the class hierarchy chart below



An important point to note is that, whichever stream type you use to read data from a file, you **must use the corresponding type of stream** for writing data to the file.

**Streams for Low-Level File Input and Output**

As mentioned already, FileInputStream and FileOutputStream are two stream classes that facilitate file access. **FileInputStream** allows us to **read in an sequence of bytes**, while **FileOutputStream** allows us to **write a sequence of bytes**.

The following sample program demonstrates the processing of a data file using only the FileInputStream and FileOutputStream classes.

**File-handling class** **FileInputStreamOutputStreamDemo**

**package** labsheet15.sampleprogram1;  
  
*//FileInputStreamOutputStreamDemo.java  
/\*This program demonstrates the creation of a file and the subsequent writing and reading of it using the FileInputStreeam and FileOutputStream classes\*/***import** javax.swing.\*;  
**import** java.io.\*;  
  
**public class** FileInputStreamOutputStreamDemo {  
  
 **public static void** main(String[] args) **throws** Exception {  
  
 File outFile = **new** File(**"demofile.data"**);  
  
 FileOutputStream outStream = **new** FileOutputStream(outFile);  
  
 **byte**[] byteArray = {10, 20, 30, 40, 50, 60, 70, 80};  
  
 outStream.write(byteArray);  
  
 outStream.close();  
  
 File inFile = **new** File(**"demofile.data"**);  
 FileInputStream inStream = **new** FileInputStream(inFile);  
  
 **int** fileSize = (**int**)inFile.length();  
 byteArray = **new byte**[fileSize];  
  
 inStream.read(byteArray);  
  
 String valuesRead=**""**;  
  
 **for**(**int** i = 0; i < fileSize; i++) {  
 valuesRead+=byteArray[i] + **" "**;  
 }  
  
 JOptionPane.*showMessageDialog*(**null**,**"Values read from the file are: "** + valuesRead, **"Output from File"**,JOptionPane.***INFORMATION\_MESSAGE***);  
  
 inStream.close();  
 }  
}

**Analysis of the FileInputStreamOutputStreamDemo class:**

● We’ve gone back to a **procedural style of programming** for this demo, with only one class and a main(). There is plenty of OO code in there alright, but no instantiable class. When it comes to file-handling for applications, you will find that, often, it is done “locally” within a **user-defined method**, and this method then gets called from certain parts of the application as required.

● Within the class definition header here

**public static void** main(String[] args) **throws** Exception {

you see the words “**throws Exception**”. I have put these words in to **ensure the application compiles**. I will leave the discussion of this until later in the lab sheet though.

● The program begins by creating a **File** object. The name of the file to be created here is called “**demofile.data**”. If this file already exists, its contents will be over-written at this point. In my case, the file is created at the **top-level of my project** (rather than within my labsheet15/sampleprogram1 package). If I wanted it there instead, I would need to specify the file as “**labsheet15/sampleprogram1/demofile.data**”.

● Now that the file exists, it can be processed. Note that **the physical file is actually not created yet**, a filename has been put aside for it but it doesn’t yet exist. As the file is currently empty, the first task is to try to write some information to it. Therefore, a FileOutputStream object is attached to the newly created File object to allow writing to the file. Once the FileOutputStream object gets linked to the File object, the physical file is actually created on disk.

● After this, a **byte array** is created and initialised with 8 values. We have never used the byte data type before, but it is a **primitive data type** like int, float, char etc. **byte** is a keyword of the Java language. As you’d expect it is **8 bits** in size and can hold values between -**127 to 128** inclusive. Its uses in Java are rather limited, but it is crucial for this particular demo and so is a good excuse to show it.

● With the byte array created, we now write the contents of this array to the file with the call to **write**(). Note that this method specifically **requires an array of byte** in its definition, so an array of int would not do.

● The last act in the first part of this demo is to call the **close**() method on the FileOutputStream object. It is **good practice to always close a stream** when it is no longer required (even though I do use it again in this program). This helps to **reduce resource usage** for a program and also helps to **improve program clarity**.

● The second part of the application begins by opening the file created earlier. Then a **FileInputStream** object is created and attached to the file.

● The line of code

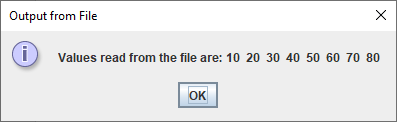
**int** fileSize = (**int**)inFile.length();

is interesting, because it gives us a chance to talk about yet another primitive data type we have hardly ever seen, the **long** data type. The method **length**() called on the File object reference returns, as a long value, the **length of the file in bytes**.

However, the next line of code needs to use this value to create the byte array that is going to store the values read from the file. You might recall that, when creating an array, the **size of it must be defined as an int**, so a long will not do (syntax error). Therefore, the long value returned by length() must be **type-cast** to an int.

● The byte array is then created and, after this, data is read from the file through the method **read**(), which takes the **array of byte** as an argument. The read() method ends up populating the array of byte with values that it has read from the file.

● Once the array has been populated, a loop is used to gather all the values in the array into a String “**list**” variable, which is then displayed on a message dialog to produce the following output:



● Following this, the FileInputStream object is closed with the call to **close**().

Copy the **sampleprogram1** package to your **labsheet15** package now. This gives you your own copy of the application. To begin with, **compile and run** the application to see its output and mess around with it a little to see its capabilities. Try to add the value 245 to the byte array. What happens? Is that what you expect? Write a **for-each loop** to replace the regular for loop I have written for processing the byte array.

**Streams for High-Level File Input and Output**

**High-level** input and output streams such as **ObjectInputStream** and **ObjectOutputStream** end up using the **lower-level** streams such as FileInputStream and FileOutputStream to help them accomplish their work of reading/writing high-level entities such as **objects** to/from files.

However, in order **to be able to write objects of a particular class to a file**, that class must implement a special Java **interface** called **Serializable**.

**Serialisation** is the process by which the state **of an object is converted into a byte stream**, which can then be written and saved to a file on disk or even sent over a network to another machine. **Deserialisation** is the **reverse process**, allowing us to read the byte stream from a file and reconstructing the original object from it.

The processes of serialisation and deserialisation are accomplished by the methods **writeObject**() and **readObject**(), called respectively on instances of ObjectOutputStream and ObjectInputStream.

So, if we have a user-defined class called Book and we wish to be able to save Book objects to a file on disk, then we would **need to make the Book class serialisable** as follows:

**public class Book implements Serializable{**

The next sample program demonstrates the creation of a serializable class, then creating some objects from the class and writing/reading them to a file on disk

**File-handling class** **SerialisationDemo**

**package** labsheet15.sampleprogram2;  
  
*//SerialisationDemo.java  
/\*This program demonstrates the creation of a file and the subsequent writing and reading of it using the ObjectOutputStream and ObjectInputStream classes to perform serialisation and deserialisation\*/***import** javax.swing.\*;  
**import** java.io.\*;  
**import** java.util.ArrayList;  
  
**public class** SerialisationDemo {  
  
 **public static void** main(String[] args) **throws** Exception {  
  
 File outFile = **new** File(**"books\_computers.data"**);  
  
 FileOutputStream outStream = **new** FileOutputStream(outFile);  
  
 Book book1 = **new** Book(**"Jaws"**,23.33,**"8978979854"**,121);  
 Book book2 = **new** Book(**"One Flew Over the Cuckoo's Nest"**,43.21,**"3453453435"**,328);  
  
 Computer comp1 = **new** Computer(**"Dell"**,**"Laptop"**,3.25,16,550.99);  
 Computer comp2 = **new** Computer(**"Apple"**,**"Desktop"**,3.85,8,854.54);  
  
 ObjectOutputStream objectOutStream = **new** ObjectOutputStream(outStream);  
  
 objectOutStream.writeObject(book1);  
 objectOutStream.writeObject(book2);  
 objectOutStream.writeObject(comp1);  
 objectOutStream.writeObject(comp2);  
  
 ArrayList<Object> mixtureOfObjects = **new** ArrayList<>();  
 mixtureOfObjects.add(**"Student"**);  
 mixtureOfObjects.add(comp1);  
 mixtureOfObjects.add(3546);  
 mixtureOfObjects.add(102.56);  
 mixtureOfObjects.add(book2);  
 mixtureOfObjects.add(**'k'**);  
  
 objectOutStream.writeObject(mixtureOfObjects);  
  
 outStream.close();  
  
 File inFile = **new** File(**"books\_computers.data"**);  
 FileInputStream inStream = **new** FileInputStream(inFile);  
  
 ObjectInputStream objectInStream = **new** ObjectInputStream(inStream);  
  
 book1 = (Book) objectInStream.readObject();  
 book2 = (Book) objectInStream.readObject();  
 comp1 = (Computer) objectInStream.readObject();  
 comp2 = (Computer) objectInStream.readObject();  
 mixtureOfObjects = (ArrayList<Object>) objectInStream.readObject();  
  
 String objectMixture=**""**;  
  
 **for**(Object o: mixtureOfObjects)  
 objectMixture += o + **"\n"**;  
  
 JOptionPane.*showMessageDialog*(**null**,**"State of standalone objects read from the file are:\n\n"** + book1 + **"\n"** +  
 book2 + **"\n"** + comp1 + **"\n"** + comp2 + **"\n\n\nThe ones from the array-list are:\n\n"** + objectMixture,  
 **"Output from File"**,JOptionPane.***INFORMATION\_MESSAGE***);  
  
 inStream.close();  
 }  
}

**Analysis of the SerialisationDemo file-handling class:**

● This program begins by creating a **File** object and then opening it. The file on disk is called “**books\_computers.data**”. A **FileOutputStream** object is created and linked to the file for writing to it, this actually **creates the physical file**. Next some **Book** and **Computer** objects are created. These instantiable classes come from Lab Sheet 4 and, like all things OO, its good to reuse stuff 😊

● Note that I had to make one small amendment to both the Book and Computer classes for this application. I had to make them both implement the **Serializable** interface as follows:

**public class** Computer **implements** Serializable {

**public class** Book **implements** Serializable {

because, otherwise, it would be **impossible to serialise** objects created from them.

● An object of class **ObjectOutputStream** is created next and this is linked to the FileOutputStream object, via a reference to the latter. This link is crucial because the FileOutputStream object is still needed to write bytes of data to the file, even though it is the ObjectOutputStream that will do the serialising.

● The code:

objectOutStream.writeObject(book1);  
 objectOutStream.writeObject(book2);  
 objectOutStream.writeObject(comp1);  
 objectOutStream.writeObject(comp2);

takes the Book and Computer objects created, serialises them and writes them to the file created earlier via the call to **writeObject**().

● After this, an **array-list** of Object is created. Of course, if we create an array-list of Object, then we can literally **place any kind of object whatsoever into the array-list**, since every class inherits from Object. So I just add a mish-mash of various kinds of object to the array-list here as you can see. The idea here is mainly to demonstrate that you **can write a collection class object to a file** just as easily as you can write standalone objects, as we did in the previous part. So, with the various objects added to the array-list, the array-list is then serialised and written to the file, before closing the file.

● The second part of the application involves the reverse process. The file is now opened and linked to a FileInputStream object, so that bytes can be read from it. An **ObjectInputStream** object is then created and linked to the FileInputStream object. It is the ObjectInputStream object that will be doing the **deserialising** via its **readObject**() method.

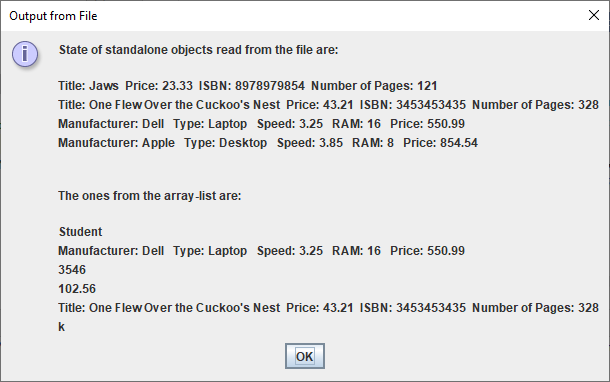
● The code:

book1 = (Book) objectInStream.readObject();  
 book2 = (Book) objectInStream.readObject();  
 comp1 = (Computer) objectInStream.readObject();  
 comp2 = (Computer) objectInStream.readObject();  
 mixtureOfObjects = (ArrayList<Object>) objectInStream.readObject();

does the job of deserialising the data coming from the file, thereby reconstructing the original objects.

Note that the **readObject**() method **returns an** **Object** (because there is no way of knowing in advance exactly what type of object is going to be read from a file). We then need to **typecast** this to the correct type of object, **so that the deserialisation can be successful**. It is important to note also that the **objects must be deserialised in exactly the same order in which they were serialised** in the first place, otherwise **runtime errors** can occur.

● After the deserialisation is completed, a **for-each** loop iterates over the contents of the array-list to build up a “list” String, to be displayed in a message dialog. After this the message dialog is displayed and contains the following:



● Finally, the FileInputStream object is closed with the call to **close**().

Copy the **sampleprogram2** package to your **labsheet15** package now. This gives you your own copy of the application. To begin with, **compile and run** the application to see its output and mess around with it a little to see its capabilities. Change the file name when reading from the file to “**books\_computers\_2.data**”. What happens when you compile? Is this what you expected? What about when you run? With the file name put back to its correct value, now try to read book2 before book1. What happens when you run? Now try to read comp1 before you read book1. What happens when you run? **Experiment** with these programs as much as you can as you will learn a lot from it.

**Exception-Handling When Dealing with Files**

As mentioned earlier, it was necessary to write the main() definition headers of the two demo programs as follows:

**public static void** main(String[] args) **throws** Exception {

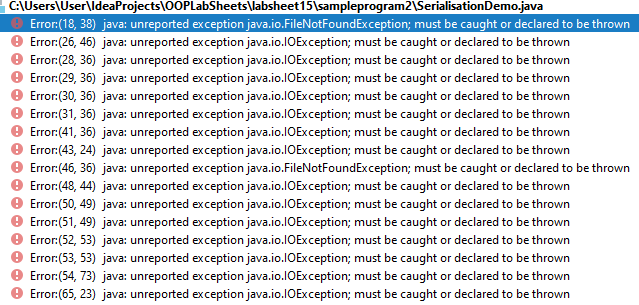
with the words “**throws Exception**”, something you won’t have seen before.

This is necessary for these applications because, when file-handling takes place, there are many things that can go wrong:

* The file you are trying to access does not exist, or has been moved in the meantime to some other location
* The file is read-only and you don’t have permission to write to it
* You have spelled the name of the file incorrectly in your application and cannot access it as a result
* The file has simply become corrupt and cannot be read from or written to

Because these types of issue are so commonplace, the designers of Java decided to **“force” the programmer to deal with** these types of potential issues directly within their file-handling applications.

If I were to omit those words “**throws Exception**” from SerialisationDemo, I would get the following at compile time:



The compiler is looking at the code, sees that there is file-handling involved from the methods being called, and because I haven’t taken any measures in my code to deal with potential runtime issues that might arise, I am given a heap of syntax error messages as you can see. They all specify “**unreported exception**” and there are two specific types mentioned – **IOException** and **FileNotFoundException**. I am told that these exceptions must be “**caught or declared to be thrown**”.

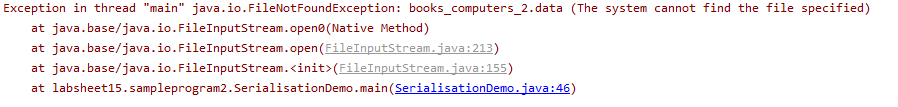
These categories of exception are referred to as “**checked exceptions**”, because the **compiler** **checks for them at compile time**. The other type are “**unchecked exceptions**” and these manifest as **runtime exceptions**, for example, when you use nextInt() to read in a numeric value, and you supply a piece of text instead, then a runtime exception occurs and you see your **program crashing**. The compiler never checks to see whether you have any code in your application to deal with the possibility of such an exception arising, so your program will compile just fine without adding anything extra. With **checked exceptions**, we **must take some kind of action** as programmers, we are forced to do so.

In the file-handling demos earlier, I took the “lazy” option by adding the words “**throws Exception**” to the main() definition header.

In effect I was saying “yes, I know that the code contained within my main() method could potentially throw up some runtime exceptions, but I am not taking responsibility for handling it here - instead, if an exception does arise, I will just let the program crash, tough luck!”.

So the Java **keyword** **throws** means “**passing the buck**”, you are “throwing the exception” out of the method that contains the code which could trigger that exception, leaving it to the Java runtime to deal with any such exception if one arises. The Java runtime will do nothing, of course, except to **display the type of exception** that occurred and **print a stack-trace** when the program crashes. The **stack-trace gives a listing of all the methods** that were called in the lead-up to the crash occurring and is very useful for **debugging** problem code.

So, for example, if I change the file name in the SerialisationDemo class to “**books\_compters\_2.data**” in the second part of the application, then at runtime I see the following when the program crashes:



I am told that a **FileNotFoundException** has occurred and the crash happened as a result of code in line 46 of the main() of my program.

The alternative to “throwing the exception” out of the main() method here is to **add event-handling code** to “**catch**” the exception should it arise. This is the recommended approach and is considered **good programming practice**.

**File Exception-Handling with try-catch**

I now alter the **SerialisationDemo** application as follows – this is within a new class called **SerialisationDemoHandlingExceptions**

**public static void** main(String[] args) {  
  
 File outFile = **new** File(**"books\_computers.data"**);  
  
 **try** {  
 FileOutputStream outStream = **new** FileOutputStream(outFile);  
  
 Book book1 = **new** Book(**"Jaws"**, 23.33, **"8978979854"**, 121);  
 Book book2 = **new** Book(**"One Flew Over the Cuckoo's Nest"**, 43.21, **"3453453435"**, 328);  
  
 Computer comp1 = **new** Computer(**"Dell"**, **"Laptop"**, 3.25, 16, 550.99);  
 Computer comp2 = **new** Computer(**"Apple"**, **"Desktop"**, 3.85, 8, 854.54);  
  
 ObjectOutputStream objectOutStream = **new** ObjectOutputStream(outStream);  
  
 objectOutStream.writeObject(book1);  
 objectOutStream.writeObject(book2);  
 objectOutStream.writeObject(comp1);  
 objectOutStream.writeObject(comp2);  
  
 ArrayList<Object> mixtureOfObjects = **new** ArrayList<>();  
  
 mixtureOfObjects.add(**"Student"**);  
 mixtureOfObjects.add(comp1);  
 mixtureOfObjects.add(3546);  
 mixtureOfObjects.add(102.56);  
 mixtureOfObjects.add(book2);  
 mixtureOfObjects.add(**'k'**);  
  
 objectOutStream.writeObject(mixtureOfObjects);  
  
 outStream.close();  
 }  
 **catch**(FileNotFoundException fnfe){  
 System.***out***.println(fnfe.getStackTrace());  
 JOptionPane.*showMessageDialog*(**null**,**"File could not be found!"**,  
 **"Problem Finding File!"**,JOptionPane.***ERROR\_MESSAGE***);  
 }  
 **catch**(IOException ioe){  
 System.***out***.println(ioe.getStackTrace());  
 JOptionPane.*showMessageDialog*(**null**,**"File could not be written!"**,  
 **"Problem Writing to File!"**,JOptionPane.***ERROR\_MESSAGE***);  
 }

File inFile = **new** File(**"books\_computers\_2.data"**);  
  
 **try** {  
 FileInputStream inStream = **new** FileInputStream(inFile);  
  
 ObjectInputStream objectInStream = **new** ObjectInputStream(inStream);  
  
 Book book1 = (Book) objectInStream.readObject();  
 Book book2 = (Book) objectInStream.readObject();  
 Computer comp1 = (Computer) objectInStream.readObject();  
 Computer comp2 = (Computer) objectInStream.readObject();  
 ArrayList<Object> mixtureOfObjects = (ArrayList<Object>) objectInStream.readObject();  
  
 String objectMixture = **""**;  
  
 **for** (Object o : mixtureOfObjects)  
 objectMixture += o + **"\n"**;  
  
 JOptionPane.*showMessageDialog*(**null**, **"State of standalone objects read from the file are:\n\n"** + book1 + **"\n"** +  
 book2 + **"\n"** + comp1 + **"\n"** + comp2 + **"\n\n\nThe ones from the array-list are:\n\n"** + objectMixture,  
 **"Output from File"**, JOptionPane.***INFORMATION\_MESSAGE***);  
  
 inStream.close();  
 }  
 **catch**(FileNotFoundException fnfe){  
 fnfe.printStackTrace();  
 JOptionPane.*showMessageDialog*(**null**,**"File could not be found!"**,  
 **"Problem Finding File!"**,JOptionPane.***ERROR\_MESSAGE***);  
 }  
 **catch**(IOException ioe){  
 ioe.printStackTrace();  
 JOptionPane.*showMessageDialog*(**null**,**"File could not be read!"**,  
 **"Problem Writing to File!"**,JOptionPane.***ERROR\_MESSAGE***);  
 } **catch** (ClassNotFoundException cnfe) {  
 cnfe.printStackTrace();

JOptionPane.*showMessageDialog*(**null**,**"Could not convert object to**

**the appropriate class!"**,**"Problem Converting Object Read**

**From File!"**,JOptionPane.***ERROR\_MESSAGE***);

}

**catch** (ClassCastException cce) {  
  *cce.printStackTrace();* JOptionPane.*showMessageDialog*(**null**,**"Could not convert the object to**

**the appropriate class!"**,**"Problem Converting**

**Object!"**,JOptionPane.***ERROR\_MESSAGE***);  
 }  
 }  
}

I have **removed** the “**throws Exception**” from the top of main() and yet all those compile time errors have now disappeared. This is because I have added **try-catch exception-handling clauses** around the parts of the code that could trigger a “checked” exception. This keeps the compiler (and me!) happy.

The **general syntax** for the try-catch clause is:

**try{**

//”try” to carry out a certain chunk of code (some of which could throw exceptions)

**}**

**catch(***ExceptionType* ex**){**

//this code executes if an exception of this particular type has been “caught”

**}**

**catch(***ExceptionType* ex**) {**

//this code executes if an exception of this particular type has been “caught”

**}**

:

: //you can have many catch clauses

:

**finally{**

//this code executes in any case, whether an exception is thrown or not

**}**

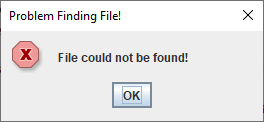
There is a **hierarchy of exception classes**, which we will look at further in the next lab sheet.

When **writing to the file**, the file may not be found, which could throw a FileNotFoundException, or it might not be possible to write to the file even if it is found, which could throw an IOException.

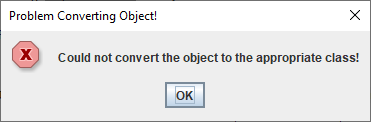
When **reading from the file**, the file may not be found, which could throw a FileNotFoundException, or it might not be possible to read from the file even if it is found, which could throw an IOException. Also, because there is **type-casting** involved in the values read from the file, a ClassNotFoundException or a ClassCastException could be thrown. ClassNotFoundException is a **checked exception** while ClassCastException is **unchecked**.

Therefore, I put in try-catch code to deal with all these possibilities. If such an exception arises at runtime now, it will be “caught” and handled by the appropriate “**catch**” clause.

So now, when it comes to attempting to access the file “books\_computers\_2.data”, the following message dialog appears, **instead of the program crashing**. In other words we “**try**” to execute a certain block of code. If an exception arises during the execution of that **try-block**, it gets “caught” and handled in a “**catch**” clause.



If the filename is correct, but I change the order of reading from the file, so that I “**try**” to read a Computer object before a Book object, then a ClassCastException is thrown at runtime, but it gets caught by this “**catch**” clause of the event-handling code and I get the following message dialog.



Note that I put in the calls to method **printStackTrace**() deliberately for **debugging/test purposes** here, as they give me the full error message on the console that would have been generated had the program been allowed to crash. They help to track down the origin of the problem and **can be removed** once the program has been fully tested.

The big difference now though is that the **program doesn’t crash at runtime** as it did before when an exception occurred. Thanks to the try-catch code, the program deals with any issues internally and so **makes the application more robust**.

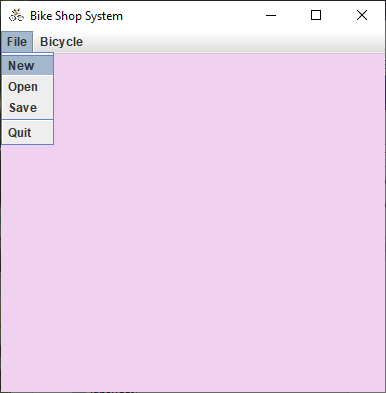
We will discuss event-handling in much more detail in the next lab sheet, but it was important to touch on it now because **you will need to handle files as part of your mini-project**. Hopefully, the little bit we covered on event-handling particular to files here will set you up for that.

Copy the **sampleprogram3** package to your **labsheet15** package now. This gives you your own copy of the amended application containing the **try-catch** code. To begin with, **compile and run** the application to see its output and mess around with it a little to see its capabilities. **Try to “break” it** as I indicated above to see if you can get it to handle some exceptions.

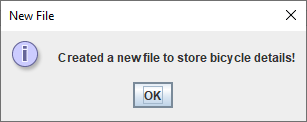
**Exercise 1**

Create a **new package** called **exercise1** within your **labsheet15** package. To this package you should **copy** the files **BicycleShopAppWithExceptionHandlingPartial.java** and **Bicycle.java** that I have within the **partialapp** package and you can also copy the image file **bike.png**.

The application is based on **Lab Sheet 14 Exercise 1**, which involved the development of a menu-based GUI

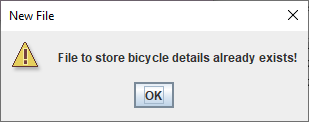


You’ll recall that the GUI was built, but there was **no functionality** behind it. Now we wish to add functionality for the **file-handling** menu-items specifically. The files given to you are incomplete and both need alterations. The application won’t compile initially because I have removed chunks of its code. Your task here is to try to use the existing code, along with my comments in **pseudo-code** form, to try to get the application fully-functional, from a file-handling point of view. So, for example, when you click on **any** of the File menu (New/Open/Save) options **for the very first time** you will see the following:

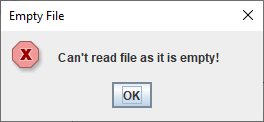


And a file called **bicycles.data** will appear (you may need to alter the file path of it depending on your own package structure).

Once the file is created, any attempt to create it again using the “New” menu-item will receive the following

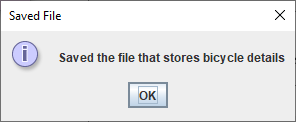


If an attempt is made to “Open” (read) from the file, and it has not been saved to yet, then the following message will issue:

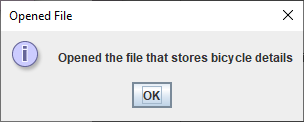


The file will only get written to when the “Save” menu-item is selected. In our case, 2 Bicycle objects get created, then added to an array-list and then this array-list should be serialised and written to the file.

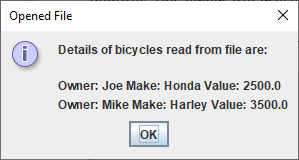
When the “Save” menu-item is selected, the following message should issue:



Now, with information saved to the file, if I select the “Open” menu-item, I receive the following (the output will depend on what values you created for your Bicycle objects of course):



followed by:



Any further saving in this case will simply overwrite the existing contents of the file (so you essentially end up with the same information unless you alter your hard-coded values).

The main goal here is to try to learn something about file-handling and exception-handling for files, to **prevent the application from crashing**.

**Text File Input and Output**

As you now know, the **FileOutputStream** class is used to **send bytes of data** representing primitive values to a file. However, this data is present in the file in a **machine-readable form**, so it does us little good to examine the file directly, as the information would make no sense.

The **FileInputStream** class is used to **read bytes of data**, representing primitive values, from a file.

In order to read the data back correctly, we **must know the order** in which the data was stored and their **associated data types**.

The classes **FileReader** and **FileWriter** are most commonly used to write text (human-readable) files, so we are dealing with **character streams** rather than byte streams.

The following sample program demonstrate the use of these classes:

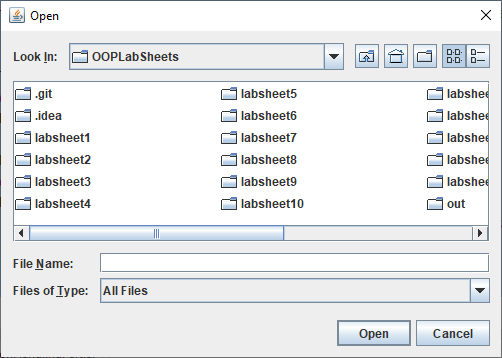
**File-handling class TextFileDemo**

package labsheet15.sampleprogram4;  
  
*//TextFileDemo.java  
/\*This file-handling application copies the contents of one text file to another. It uses  
\* a GUI component called a JFileChooser to allow the user to pick the file they wish to  
\* copy\*/*import javax.swing.\*;  
import java.io.\*;  
  
public class TextFileDemo {  
  
 public static void main(String args[]) {  
  
 JFileChooser fileChooser = new JFileChooser(".");  
 int status = fileChooser.showOpenDialog(null);  
  
 File selectedFile = null;  
  
 if(status == JFileChooser.*APPROVE\_OPTION*)  
 selectedFile = fileChooser.getSelectedFile();  
  
 try{  
 if(selectedFile!=null) {  
 FileReader in = new FileReader(selectedFile);  
  
 FileWriter out = new FileWriter("output.txt");  
  
 int c;  
  
 while ((c = in.read()) != -1)   
 out.write(c);  
  
 JOptionPane.*showMessageDialog*(null, "Data now copied from " + selectedFile.getName() + "

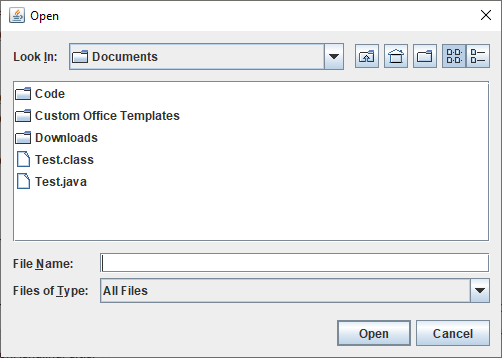
to " +"output.txt", "File Copied!", JOptionPane.*INFORMATION\_MESSAGE*);  
  
 in.close();  
 out.close();  
 }  
 else  
 JOptionPane.*showMessageDialog*(null, "No file was selected!",  
 "No File Selected!", JOptionPane.*ERROR\_MESSAGE*);  
  
 } catch (FileNotFoundException fnfe) {  
 System.*out*.println(fnfe.getStackTrace());  
 JOptionPane.*showMessageDialog*(null, "File could not be found!",  
 "Problem Finding File!", JOptionPane.*ERROR\_MESSAGE*);  
 } catch (IOException ioe) {  
 System.*out*.println(ioe.getStackTrace());  
 JOptionPane.*showMessageDialog*(null, "File could not be read/written!",  
 "Problem Reading/Writing to File!", JOptionPane.*ERROR\_MESSAGE*);  
 }  
 }  
}

**Analysis of the TextFileDemo file-handling class:**

● The main() begins by creating an object from a GUI class we haven’t used before, called **JFileChooser**. This class is useful because it presents us, at runtime, with a dialog that **enables us to traverse the file system visually** in order to manually select a file that we want to choose for our application, a bit like File Explorer in Windows. It appears as follows as runtime:



The constructor method takes the name of the starting directory. I put in “.” which represents the **current working directory**, for me that resolves to my OOPLabSheets folder on the C: drive of my machine. For you, it will depend on where you are running the application from. If you change the directory from “.” to something else, for example, C:/users/documents it would appear as follows instead (yours would look different of course, depending on what files and folders you have within your Documents folder).

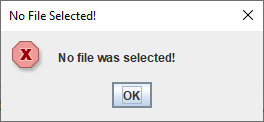


● The dialogs you see above are as a result of the call to the method **showOpenDialog**() on the JFileChooser object reference. The user can interact with this dialog and navigate through the file system to select a file (you could **create a folder** also if you wished, just like you could with File Explorer). If the user does select a file, then this method will return the integer value of the constant **JFileChooser.APPROVE\_OPTION**. Otherwise it would return a different constant (either **JFileChooser.CANCEL\_OPTION** or **JFileChooser.ERROR\_OPTION**).

The value returned by the method is stored in the **status** variable and this is then compared to **JFileChooser.APPROVE\_OPTION**. If they match, then the method **getSelectedFile**() is called on the reference to the JFileChooser object to return the **File** object selected by the user.

● At this point a **try-catch** structure surrounds code where **file-handling** is involved and which could potentially throw up **runtime errors**. As mentioned before, the exception types listed in the “**catch**” clauses are **checked exceptions** that we are forced to deal with within our application by the Java system. Our program could not compile unless we deal with them in some way, and using try-catch is the best approach. Can you recall what the other alternative would be?

● The first thing that happens within the try clause is a test to determine whether the variable selectedFile is null. If it is then it means that the user never selected a file from the JFileChooser and so we cannot proceed with the rest of the application (because we are trying to copy the contents of one file to another file) and so an error message dialog appears as follows:



and the program terminates at that point. However, if they do select a file, then the value of selectedFile will not be null (it will be referencing some File object) and we can go on.

● Now that we know a file was selected, a **FileReader** object is created and linked to the File object. It opens the file for reading.

● Next a **FileWriter** object is created and this is linked to a File object that will have the name “**output.txt**” in my case. By default this file will be created just beneath my “OOPLabSheets” folder, but you could get it created anywhere you like by giving it a specific location. Once the link is made the file is opened for writing.

● The code

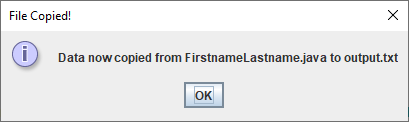
int c;  
  
while ((c = in.read()) != -1)  
 out.write(c);

will read from the File object linked to the **FileReader**, one character at a time, using the **read**() method, until the **end of the file** has been reached (signalled by the value -1). For every character that is read from the File referenced by **in**, it will immediately be written to “**output.txt**” via the call to **write**() on the **FileWriter** object reference **out**.

In this way, the entire contents of the file selected by the user will be copied to the file “output.txt”.

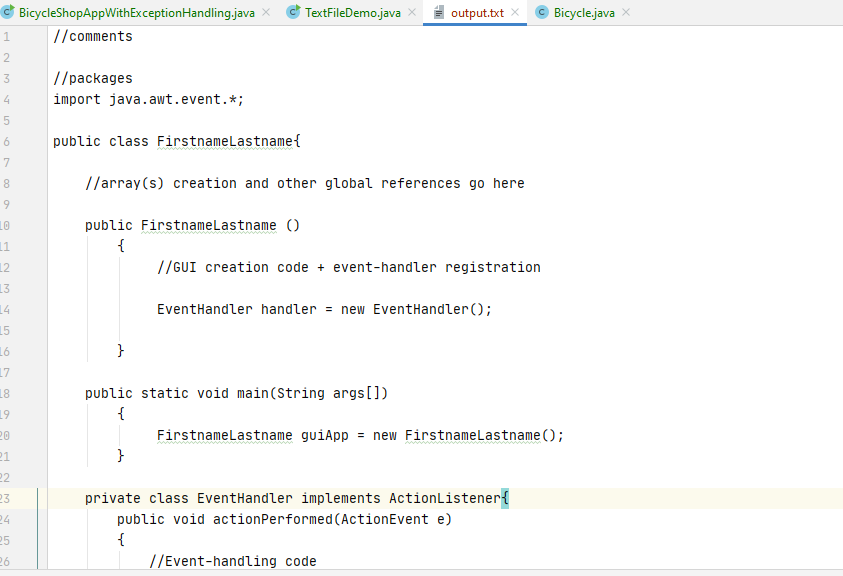
● Once the loop finishes, a message dialog is issued to confirm that the file is now copied and it indicates the source and destination files. The name of the source file is obtained via the call to method **getName**() on the **selectedFile** reference.

In my case, I got the following:



● The try clause finishes by **closing** both the FileReader and FileWriter streams. Can you recall the reason for doing this?

Once the application terminates and I open the file output.txt, I see the following:



proving the copy was successful.

Copy the **sampleprogram4** package to your **labsheet15** package now. This gives you your own copy of the application. To begin with, **compile and run** the application to see its output and mess around with it a little to see its capabilities.

**Exercise 2**

Create a **new package** called **exercise2** within your **labsheet15** package. Create a Java class called **RandomNumberFileHandling** with a main() that generates a set of 100 random whole numbers between 1 and 500 and writes them, one per line, to a text file called **randomnumbers.txt**. Note that there is no version of the **write**() method that allows you to write integers – you can only write characters and String values. Therefore, you will need to convert each randomly generated number to a String first before you can successfully write it to the file. See if you can recall how to do this.

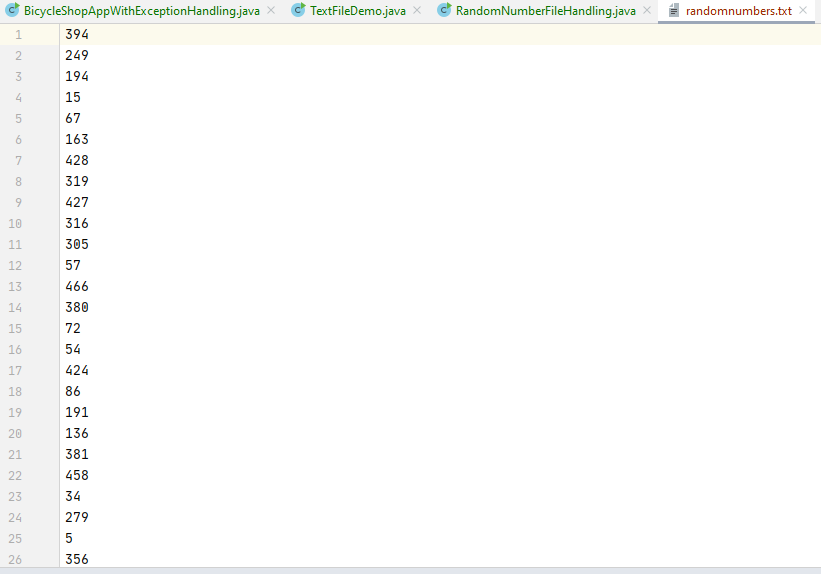
Then, once the writing to this file is completed, the program should read from the text file using the **Scanner** class, one number at a time using a **while** loop – you will use **nextLine**() here and then convert the value read to an integer. The numbers read from the file should be added to an integer array, one by one, using the same **while** loop.

Once the array is populated, it should then be processed by a **for-each** loop which will allow the program to determine the **average** of the numbers, along with the **highest** and **lowest** numbers. The for-each loop should also contain code to build-up a string containing all the random numbers, that will be displayed later on a text-area in a message dialog.

Once the for-each loop completes, the set of numbers should be displayed neatly aligned on a text-area within a message dialog, with 10 numbers per line, along with the **average** of the numbers, to **2 decimal places** and the **highest** and **lowest** numbers.

Your program will run similar to the following:

A file containing the 100 randomly generated numbers will be created, mine looks as follows, but these numbers will be different on every run.



Then the numbers are read from the file using Scanner, processed, and then the results appear as follows in a text-area on a message dialog:

