The contents of this document represent and go over the topics covered in lecture on 4/4 and 4/6 Use this to review the content as you please, also be sure not to use these in place of lecture as these documents will only cover lecture highlights and important details and do not represent the full scope of what you may be tested on. As always if you have any questions ask on ED!

Lecture 21 - 4/4/2023

Reading and writing to files in Java is a process that brings together lots of different concepts.

In order to begin handling files in the first place you will need to import the Scanner class (this you know how to do) you also need to import all the stuff relating to files, which is in the java.io package. So you'd need to import java.io.*; remember * here acts as a wildcard.

We are introduced to the File object type, which takes in a file path in the form as a string as the sole argument. Here is an example:

```
File myFile = new File("<pathToFile>");
```

Coupling this with a Scanner we now have a means to parse a file:

```
Scanner sc = new Scanner(myFile);
```

In order to write to a file we need to use an object known as a PrintWriter:

```
PrintWriter output = new PrintWriter("<pathToFile>");
```

We can iterate through the contents of the file that have been loaded into the scanner using the .next() method on a Scanner object, combining this with the hasNext() method we can form a loop to continually iterate through until we can't anymore:

```
while(sc.hasNext()){
      //some code dealing with the next() method
 }
```

To print to the output file, we use the printwriter's println method so in our case:

```
output.println(<whatYouWantToPrint>);
```

You must close the PrintWriter after you finish printing for the output to be

saved.
We must also finally talk in depth about exceptions. There are two kinds of exceptions, checked, and unchecked exceptions. You must provide a way to handle checked exceptions or the compiler will complain to you. Exceptions dealing with files, FileNotFoundException, IOException etc are all checked exceptions. In order to properly deal with this we have two methods.

The first is writing "throws someException" in our method signature as displayed in ExampleA.java in Lecture 21 on codio. This clause tells the compiler that we are aware that our method could throw someException and we are willing to take that risk. The compiler accepts this and decides not to bother you.

The second way is by writing a try-catch block, what this effectively means is try the code within the try block, if an exception is thrown that we intend to catch we move to the respective catch block and execute whatever code is there.

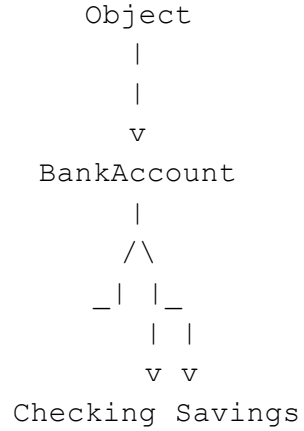
From ExampleB.java:

```
        try{
            File inFile = new File(args[0]);
            Scanner input = new Scanner(inFile);
            String word;
            PrintWriter output = new PrintWriter(args[1]);
            while(input.hasNext())
            {
                word=input.nextLine();
                output.println(word);
             }
            output.close();
        }
        catch(FileNotFoundException e){
            System.out.println("Try again with correct input file name");
            System.out.println(e);
        }
```

We attempt to run the code in the try block, if we get a FileNotFoundException we move to the catch block and execute the code written in there. You are able to catch multiple exceptions as well with different catch blocks, this can be seen in ExampleC.java and ExampleD.java
Lecture 22 - 4/6/2023

Today's lecture is the start of a portion of the course that gave me literal hell when I was encountering it for the first time. Anecdotes aside, today we

began to talk about inheritance. In Java there can be super classes or subclasses of another class. Every class can only have 1 parent. Take the following hierarchy diagram:



These are the classes that can be found in Lecture 22 on Codio. Object is the highest class in the hierarchy all Java Classes implicitly inherit from Object, the toString() method that you overrode in Card.java and Deck.java is such an example. Speaking of the word *overrode*, that is a new word! You already know about method overloading, but now we need to define method overriding. To override a method is to reimplement a method that you inherited from a superclass. The method signature stays the same in this case as well.

To establish the relationship between classes we use the "extends" keyword in our class definition. Note that you can only extend one class since Java classes can only have 1 parent.

```
public class SubClass extends SuperClass {
      //instance variables

      public SubClass(<parameterList>){
            //TODO implement below
      }
      //methods and other stuff
}
```

When we establish a relationship between classes like this we tend to say that "SubClass **IS A** SuperClass" For example, a checking account is a bank account, same for savings account. The inverse is not true though, a BankAccount is NOT a SavingsAccount and so on. This will be important when we discuss Polymorphism.

When we establish a class relationship like this, we must pay extra attention to the constructor of the subclass, if we wish to properly initialize the

instance variables that we inherit, we will need to invoke the constructor of the super class, and it has to be the **first** thing we do so consider an implementation of the constructor above:

```
public SubClass(<parameterList>){
      super(<necessaryParameters>);
      //other statements
}
```

We can also use the super keyword to access methods from the superclass. In the Lecture 22 codio files there is an example where we write **super.withdraw(<x>)** This statement called the withdraw method from the BankAccount.

Finally we can talk about Polymorphism, which means "many forms" This is the pinnacle of this inheritance stuff, if we had multiple types of BankAccounts and we wanted a method to update all of them, we can simply use the BankAccount type since it is the super class of all subsequent classes that are versions of a BankAccount. This is also true when it comes to declaring Objects.

```
BankAccount a = new BankAccount("Griffin");
BankAccount b = new Checking("Gabbie", 0.5);
BankAccount c = new Savings("Cannon", 1.2);
```

This is great and all but we must be careful of the issues with Compile Time Polymorphism, or Static Polymorphism.

At Compile Time Java only sees the declared type, in all the above cases this would be BankAccount but any attempt to call methods from the instantiated type will not compile. So the following are examples of statements that do not compile:

1. b.monthly();
2. c.daily();

But all the following do compile:

1. a.withdraw(<x>);
2. b.withdraw(<x>);

Finally we can talk about Runtime Polymorphism or Dynamic Polymorphism. With the two above statements, they compile but at runtime they do different things, this is because the instantiated types are different thus a will call BankAccount's version of withdraw() and b will call Checking's withdraw. Interfaces were mentioned but not talked about in detail so I will reserve them for next time.