Lecture 21 - 4/2/2024

We started the lecture by discussing a new type of try block, specifically the try with resources statement. You simply put resources like Scanners and PrintWriters inside parens following the try reserved word and they will automatically be closed once the try block finishes:

```
 try(Scanner input = new Scanner(inFile);
     PrintWriter output = new PrintWriter(args[1])){
   //code for try
 }
```

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:

```
        Object
          |
          |
          v
     BankAccount
          |
         /\
       _| |_
       |    |
       v    v
  Checking   Savings
```

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("Emily", 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.

Lecture 22 - 4/4/2024

This lecture consisted of primarily review with some extension on our discussion of exceptions primarily with regard to two key aspects:

1. Creation of Custom Exceptions

2. Exception Propagation

We can actually make custom exceptions that derive from other exceptions to serve our own purposes, we saw in lecture the OverdrawException that we made to handle the case where a person attempts to withdraw more money than they have within their BankAccount:

It is important to analyze so I will paste it in its entirety here but obviously this snippet was originally written by Professor Cannon:

```
public class OverdrawException extends IllegalArgumentException{

    public OverdrawException(String m){
        super(m);
    }

    public OverdrawException() {
       super("Dude, you need money");
    }
}
```

When an exception is thrown there is a message that is displayed to the user, in this example we allow for both a custom message as well as a default message if none is provided. Just like instantiating objects we can do the same with these Exception objects too with:

```
OverdrawException e = new OverdrawException(“WOMP WOMP”);
```

We can then launch the exception at the user by writing the following:

```
throw e;
```

Trivally, we can combine the two lines and do:

```
throw new OverdrawException(“WOMP WOMP”);
```

Also it doesn’t really matter which Exception we use as the parent since they all act the same but it is convention to use the one that is most closely related to the new exception that you are writing.

We also talked about exception propagation, consider the following scenario:

```
main() -> methodA() -> methodB() -> methodC()
```

What this is saying is that main calls methodA which calls methodB which calls methodC.

Say an exception is thrown in methodC() we have the following options in terms of resolution:

1. Handle immediately
2. Handle later
3. Handle never

While you may think handling now is the best option, it is actually better to procrastinate and handle the exception later because there may be more valuable information lower in the call stack that will allow us to handle the exception in a more appropriate manner. So say we wish to handle the exception in methodA then:

```
main() -> methodA() -> methodB() throws Exception -> methodC()throws Exception
```

You will need to acknowledge the exception (required if checked, conventionally if unchecked) in all higher method calls that proceed the method that handles the exception.