COP 3330 – Object Oriented Programming in Java
Creating Exceptions for Your Own Classes

Creating Exceptions

On occasion, when you create your own class, there might be an operation a user might try that creates some sort of issue, so that the operation can't successfully be executed. In our past examples, we tried to "catch" these issues outside of the class and handle them gracefully.

One of these examples was our AddressBook program, which used an array of size 10 to store Contact objects. If a user attempted to add an eleventh contact, we gave the user the canAdd method so that they wouldn't try to add an eleventh contact. Also, if the user tried anyway, we didn't add the contact and returned false from the addContact method to indicate that the add failed.

An alternative way to handle this is to create our own Exception. This tends to be very easy to do since all exceptions inherit from Exception and don't have any "actual code." For this particular situation, we'll create the FullAddressBookException. Here is the full code for it:

```
public class FullAddressBookException extends Exception {
    public FullAddressBookException(String s) {
        super(s);
    }
}
```

Now, if there's a situation where the user's code could create this issue, we can throw this exception by calling the constructor for FullAddressBookException from that method, passing the constructor a string with specific information (if we choose).

In the previous lecture, we learned that we can choose to throw an Exception and not handle it. For this example, the file "AddressBook.java" executes this idea. In order to not handle the exception, we must report that we might throw the exception. In the posted code, the main method in the AddressBook class calls the addContact method. Since the addContact method can throw the FullAddressBookException, we must simply report that we might throw it from main also. Here is how the declaration of main changes:

```
public static void main(String[] args) throws FullAddressBookException {
    // Main method code here
}
```

In the main method, if we ever continue adding names to the AddressBook until we try to add the 11^{th} name, a run-time error will occur with something similar to the following printing out:

```
Exception in thread "main" FullAddressBookException: Friend Quota Reached!
        at AddressBook.addContact(AddressBook.java:23)
        at AddressBook.main(AddressBook.java:86)
```

Note: I've created many versions of this as I was editing for these notes, so the exact line numbers might differ from the specific version posted online.

Handling the Exception

The alternative, in the main method to throwing the FullAddressBookException would be to catch and handle it gracefully. To do this, we must call addContact in a try block with a matching catch block.

In the AddressBook.java file, we simply call the following line of code:

```
blackbook.addContact(new Contact(name,age,number,mon,day));
```

Here is the modification that handles catching the FullAddressBookException in a separate class, AddressBook2:

```
try {
    blackbook.addContact(new Contact(name,age,number,mon,day));
}

catch (FullAddressBookException e) {
    System.out.println("Can't add any more friends right now.");
}
```

Just like the previous lecture, if there's a line of code that could cause an Exception, we try it in a try clause, and then catch the appropriate Exception type in the catch clause. Notice that here, the String object from the FullAddressBookException object is not used at all. If we had wanted, we could have utilized the object e by printing out its stack trace inside of the catch block.

```
e.printStackTrace();
```

When we run the AddressBook2.java's main method and attempt to add an eleventh contact, the program simply prints out:

```
Can't add any more friends right now.
```

and proceeds to prompt the user with the main menu again. Printing the address book right afterwards will show the same 10 previous contacts stored.

Complex Number Example

One operation not allowed with Complex Numbers is a division by 0+0i operation. In this example we write code for a ComplexNumber class, including a divide method. This method will throw the DivideByZeroException we create.

We only throw this exception if the number we're dividing by has a magnitude squared less than 1e-9 (ten to the negative ninth power). Here are the relevant methods from the ComplexNumber class that allows us to include this Exception:

```
public double magSq() {
    return a*a + b*b;
}

public ComplexNumber multiply(ComplexNumber other) {
    double real = a*other.a - b*other.b;
    double img = a*other.b + b*other.a;
    return new ComplexNumber(real, img);
}

public ComplexNumber conjugate() {
    return new ComplexNumber(a, -b);
}

public ComplexNumber divide(ComplexNumber other) throws DivideByZeroException{

    ComplexNumber term = other.conjugate();
    ComplexNumber num = this.multiply(term);

    double den = term.magSq();

    if (Math.abs(den) < 1e-9)
        throw new DivideByZeroException("You can not divide by 0+0i.");

    return new ComplexNumber(num.a/den, num.b/den);
}
```

The magSq() method just returns the square of the magnitude of this ComplexNumber. multiply is included since it's used in division, which is also true of the method conjugate(). Finally, in the divide method, we start making intermediate computations that we need to compute the division. After the first two lines, the last critical number we need is the magnitude squared of term (or other). If this number is too small, then we consider it to be 0 and must throw the appropriate exception. In this code we go ahead and add to the stack trace that you can't divide by 0+0i.

If we make it past this line of code, we're safe to generate the resultant object to return.

Now, we must see how we'll handle this in our main method in the class TestComplexNumber.java.

In this class, we can safely carry out addition, subtraction and multiplication of the two user inputted ComplexNumbers. But instead of automatically doing division, we try it out in a try block and catch the DivideByZeroException as follows:

```
try {
    ComplexNumber div = n1.divide(n2);
    System.out.println(n1+" / "+n2+" = "+div);
}

catch (DivideByZeroException e) {
    System.out.println("Sorry, you can not divide by 0.");
}
```

If we succeed in trying the division, then we can move onto printing out the result, which we do in the try clause. Otherwise, we catch the error and just print out that we can't divide by 0 for the user and print no result for the division.

As can be seen, Exceptions are the formal method preferred for Java to deal with "incorrect" method calls, though, all semester, we've been dealing with some of these issues informally in alternative ways.