PART A

File I/O [7 points]

We have not explicitly covered File I/O in class so you will have to read ahead to answer the questions below.

1) [2 points] A stream is an object that allows for the flow of data between your program and some I/O device or some file. What is your opinion on this statement? Explain your answer.

> ANS: I think this statement accurately describes what a stream is. Streams act as intermediaries between your program and the actual I/O devices.

2) [2 points] Every input file and every output file used by your program has only one name which is the same name used by the operating system. Is this an accurate statement? Discuss your perspective on this.

> ANS: This is not an accurate statement. When pass a file name to a java constructor for a stream you are not giving java an identifier but a string corresponding to a file name, which has no special meaning in java so you can use any name for input and output files allowed by your OS.

3) [1 point] The FileNotFoundException is a descendant of the class?

[Hint: Refer to the Java API Documentation from Oracle]

> ANS: The IOException Class

4) [2 points] When your program finishes writing to a file, it should close the stream connected to that file. Why is this necessary?

> ANS: so the system releases any resources used to connect the stream to the file and does any other housekeeping that is needed. Even though Java implicitly closes files when you don’t do it yourself, if you don’t close the stream and your program end abnormally it could damage the file and any data that was to be written will no be leaving the file incomplete and if your program writes to a file and later reads from the same file, it must close the file after it is through writing to the file and then reopen the file for reading.

Error Handling [10 Points]

Task: Handle the exceptions below. You must copy and paste the code below into an appropriate IDE for this task. After you have handled this exception, copy the code back into this document for submission.

5) FileNotFoundException

```
package com.icp.exceptions;


import java.io.File;

import java.io.FileReader;


public class CheckedExceptions

{

public static void main(String args[])

{

try{

        File file = new File("/home/icp/file.txt");

        FileReader fileReader = new FileReader(file);
}

// attempt to open the file denoted by a specified pathname has failed

Catch (FileNotFoundException e){

        System.out.println("FileNotFoundException : Error opening the file”);

         System.exit(0);

}

}

}
```

6) NumberFormatException

```
package com.icp.exceptions;


public class UnCheckedExceptions
```

```
{
public static void main(String args[])
{
// "Number" is not a integer, it is string
//NumberFormatException
try {
        int number = Integer.parseInt ("Number");
        System.out.println(number);
}
// Number" is not a integer, it is string
catch (NumberFormatException nfe) {
        System.out.println("NumberFormat Exception: invalid input string");
}
}
```

PRACTICE [0 POINTS]

Practice 1: Programmer-Defined Error Handling

Creating an exception class

To create a custom exception class, you just define a class that extends one of the classes in the Java exception hierarchy. Usually, you extend Exception to create a custom checked exception. Suppose that you are developing a class that retrieves product data from a file or database, and you want methods that encounter I/O errors to throw a custom exception rather than the generic IOException that is provided in the Java API. You can do that by creating a class that extends the Exception class. Unfortunately, constructors are not considered to be class members, so they are not inherited when you extend a class. As a result, the ProductDataException has only a default constructor. The Exception class itself and most other exception classes have a constructor that lets you pass a string message that is stored with the exception and can be retrieved via the getMessage method. In addition to the use of constructors for messaging, there is a need to override the toString method.

}

Thus, you want to add this constructor to your class, which means that you want to add an explicit default constructor too. So now the ProductDataException class looks like this:

```
public class ProductDataException extends Exception{

        public ProductDataException{ }

        public ProductDataException(String message){super(message); }

}
```

Although it's possible to do so, adding fields or methods to a custom exception class is unusual.

Throwing a custom exception

As for any exception, you use a throw statement to throw a custom exception. You usually code this throw statement in the midst of a catch clause that catches some other, more generic exception. Here's a method that retrieves product data from a file and throws a ProductDataException if an IOException occurs:

```
public class ProductDB

{

        public static Product getProduct(String code)

                throws ProductDataException

                {

                        try{

                                Product p;

                                // code that gets the product from a file

                                // and might throw an IOException

                                p = new Product();

                                return p;

                        }

                        catch (IOException e){

                                throw new ProductDataException(

                                "An IO error occurred.");

                        }

                 }

}
```

Here is some code that calls the getProduct method and catches the exception:

```
try{
        Product p = ProductDB.getProduct(productCode);
}
catch (ProductDataException pde){
        System.out.println(pde.getMessage());}
```

Here the message is simply displayed on the console if a ProductDataException is thrown. In an actual program, you want to log the error, inform the user, and figure out how to continue the program gracefully even though this data exception has occurred.

PART B - Challenge

Programmer-Defined Error Handling [18 points]

The following exercises are intended to introduce you to programmer-defined error handling in Java.

Task: Referring to the practical above, identify one mutator method from your running examples (Food or Music) and implement a programmer-defined exception to handle a potential/possible input from the user that can result in an error.

ANS

```
public class InvalidPriceException extends Exception {
    public InvalidPriceException() {}
    public InvalidPriceException(String message){
        super(message);
    }

}
public void setPrice(double price) {
    try {
        if (price <= 0)
            throw new InvalidPriceException();
        this.price = price;

    }
    catch (InvalidPriceException ipe){
        System.out.println(ipe.getMessage());
    }
}
```