Activity 5-1

Exception Handling with Array

1. Write a program to find the average of the elements in a double array.
2. Prompt user to enter the array size, **with InputMismatchException handled,** till the array size is a valid No.
3. Declare and create a double array with size decided by user in step 2.
4. Write a method that finds the average in an array of floating-point values with header: (Please use for loop, then do while, then while loop to design it)

   public static double avgArry(double[ ] a)

5. Prompt user to enter the array element value one by one using loop **with Exception handled**
6. Call method avgArry(double[ ] a) of Activity 5-2 to calculate the average of all the elements' values of the above array , then print out the average value
7. Keep this program.

Activity 5-2

1. Below is a program that reads in a text file “a.text”’
2. It is currently having errors as File IO operations are checked exceptions.
3. Edit the program to handle FileNotFoundException in the main().

```
public class Test {
    public static void main(String[] args) {
        String s = "a.txt";
        readTextFile(s);
    }

    public static void readTextFile(String a) {
        String s;
        File myFile = new File(a);
        Scanner sc = new Scanner(myFile);
        while (sc.hasNextLine()) {          // loop to read entire file
            s = sc.nextLine();
            System.out.println(s);

        }
    }
}
```

Activity 5-3

House Pet

There is an abstract class **HousePet**. It has 2 abstract methods:

```
public abstract class HousePet
{
  protected String name, favoriteFood, owner;

  public HousePet(){
    //We'll name all of our pets Pooky initially.
    name = "Pooky";

    //We'll assume Donna owns all of the pets.
    owner = "Donna";

    //We'll assume all of our pets like cookies.
    favoriteFood = "cookies";
  }

  //Here is our overloaded constructor.
   public HousePet(String n, String o, String ff){
     name = n;
     favoriteFood = ff;
     owner = o;
  }

  //These abstract methods must be overidden in the subclasses
  public abstract String where_I_Sleep();
  public abstract String how_I_Move();
  public void setName(String n) { name = n; }
  public void setFavoriteFood(String ff) { favoriteFood = ff; }
  public void setOwner(String o) { owner = o; }
  public String toString(){   {
  String output = "I am " + name + " a house pet. "
     +"\nMy favorite food is " + favoriteFood
                         +".\nMy owner is " + owner +".";
     return output;
  }
}
```

An incomplete **Dog** class is provided as follows:

```java
public class Dog extends HousePet
{
   protected int numberOfWalksPerDay;

   public Dog(){
      //This calls HousePet() automatically.
      numberOfWalksPerDay = 2;
   }

   public Dog(String n, String o, String ff, int numWalks){
      //We must explicitly call the HousePet() overloaded
      //constructor, passing it the name, owner, and food info.
      super(n,o,ff);
      numberOfWalksPerDay = numWalks;
   }

   /*Here are the two methods that are abstract in the superclass, which
   are overriden here, thus making Dog a complete class*/

   public String where_I_Sleep(){
   /*add codes here to implement the method—describe how a Dog sleep*/
   }

   public String how_I_Move(){
   /*add codes here to implement the method—describe how a Dog move*/
   }

   public String toString(){
       String output = super.toString();
       /*modify coded here to override toString method---provide
       complete description of a Dog*/
       return output;
   }
}
```

Add codes to complete the code for class Dog and override the toString() method.

Write a Java Application to create object of Dog, then call Dog's toString() method to print out the description of a dog.

Activity 5-4

1. Below is a simple class VendingMachine.

```
class VendingMachine {
   String type;
   VendingMachine(String type) {
     this.type = type;
   }
}
```

2. There are only 2 types of vending machines.
   - "***Coin Paying Only***" (accepts coins only)
   - "***Coin Note Paying***" (accepts coins and notes)

3. The ***printInstruction()*** method is responsible for printing the respective instructions based on the type of vending machine.

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        HashMap<Integer,VendingMachine> vmGroup = new HashMap<>();
        VendingMachine v1 = new VendingMachine("Coin Paying Only");
        vmGroup.put(1, v1);
        VendingMachine v2 = new VendingMachine("Coin Note Paying");
        vmGroup.put(2, v2);

        for (int i=1; i<=vmGroup.size(); i++)
           printInstruction(vmGroup.get(i));
   }

   public static void printInstruction (VendingMachine v) {
        if (v.type.equals("Coin Paying Only")) {
           System.out.println("This machine accepts coins only.");
           System.out.println("Drop in coins.");
           System.out.println("Select item.");
           System.out.println("Press GO button.");
        }
        else if (v.type.equals("Coin Note Paying")) {
           System.out.println("This machine accepts coins and notes.");
           System.out.println("Drop in coins and insert notes.");
           System.out.println("Select item.");
           System.out.println("Press GO button.");
        }
   }
}
```

4. This is fine if the program is small and has only 1 method which runs the logic with if-else structures based on the ***type*** of vending machine. If the application has multiple code that need to run the similar ***if-else*** structure based on the ***type*** of vending machine, and constantly needs to handle new types of vending machines, then these existing code need to be changed to accommodate the new type.

5. A better design is to start with an interface (or abstract class) ***VendingMachine*** with an abstract method ***printGuide():***

```
interface VendingMachine {
   public void printGuide();

}
```

6. Next, create a class for each type of vending machine, ***CoinVendingMachine*** and ***CoinNoteVendingMachine.*** These classes will implement the interface ***VendingMachine***. Hence, each of these classes will provide the logic (i.e. the instructions) for the respective ***printGuide()*** method.

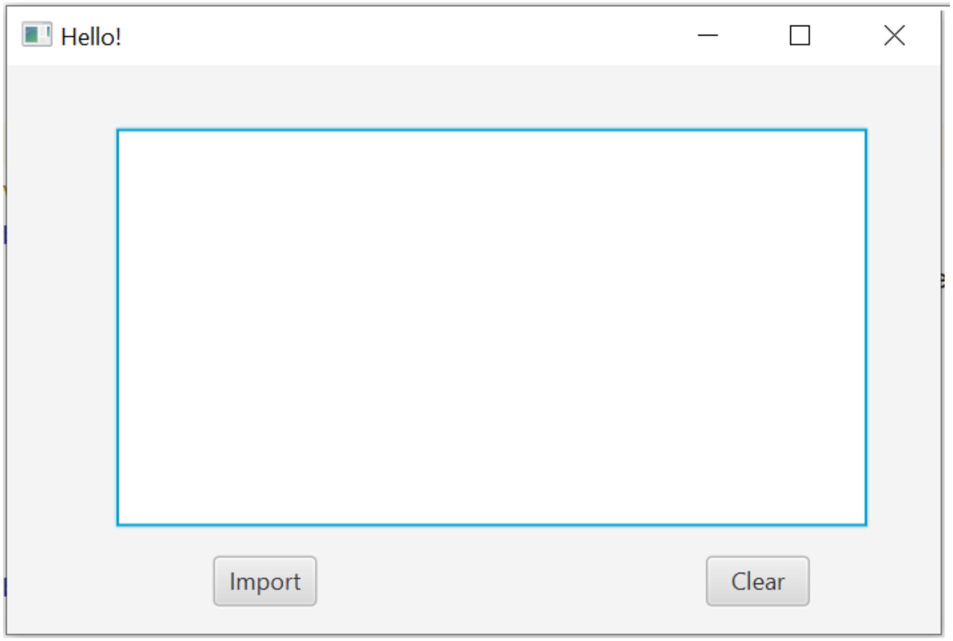
7. With this design, adding a new type of machine means adding a new class implementing the same interface without changing any of the existing class nor the **printInstruction()** and other methods.

```
public class Test {
   public static void main(String[] args) {
       HashMap<Integer,VendingMachine> vmGroup = new HashMap<>();

       CoinVendingMachine v1 = new CoinVendingMachine();
       vmGroup.put(1, v1);
       CoinNoteVendingMachine v2 = new CoinNoteVendingMachine();
       vmGroup.put(2, v2);

       for (int i=1; i<=vmGroup.size(); i++)
          printInstruction(vmGroup.get(i));
   }

   public static void printInstruction (VendingMachine v) {
        v.printGuide();

   }
}
```

8. Complete the two classes ***CoinVendingMachine*** and ***CoinNoteVendingMachine.***

Activity 5-5

1. Create a JavaFX program to have a GUI with a ListView and 2 buttons.
2. Clicking the Import button will add the list of schools in “a.text” into the ListView. (Please refer to Activity 5-2 for text file reading)



3. Clicking the Clear button will clear the ListView.
4. Sample “a.text”:

```
School Electrical and Electronic Engineering
School of Computing
School of Chemical and Life Sciences
School of Mechanical Engineering
School of Business
```

5. The following statement add a String “School is fun” into the ListView (id: schoolList)

```
schoolList.getItems().add(“School is fun”);
```