Name _____ Period _____

Dixie Cup Arrays

Your Tasks (Mark these off as you go)

- ☐ Define key vocabulary
- ☐ Write the *DixieCup* class
- ☐ Write an overloaded constructor
- ☐ Write the *setIsFull* method
- ☐ Write the *addItem* method
- ☐ Write the *getItems* method
- ☐ Write the *removeItem* method
- ☐ Write the *DixieCupMaker* class
- ☐ Add items to your DixieCups
- ☐ Remove items from your DixieCups
- ☐ Receive credit for this lab guide

☐ Define key vocabulary

Constructor overloading

Method overloading

Driver class

□ Write the *DixieCup* class

We have previously learned that classes in java allow us to model other data types that are not built into the programming language and create as many subsequent objects of the class as needed. As a review, let's consider, how you might model an Element on the periodic table. An element has the following characteristics: atomic number, mass, and a symbol. Elements can also be defined as metals or nonmetals.

A class that could be used to model an element is illustrated below,

```
public class Element{

    private int atomicNumber;
    private double atomicMass;
    private String symbol;
    private boolean isMetal;

    public Element(int atomicNumber, double atomicMass, String symbol, boolean isMetal){
        this.atomicNumber = atomicNumber;
        this.atomicMass = atomicMass;
        this.symbol = symbol;
        this.isMetal = isMetal;
    }

}
```


In the above example the following are referred to as *instance variables*. Notice the instance variables are (1) not declared inside a method and (2) are declared as private.

```
    private int atomicNumber;
    private double atomicMass;
    private String symbol;
    private boolean isMetal;
```

The next block of code is referred to as the *constructor.* Notice the constructor takes the same name as the class. In our example the constructor also contains the parameters: *atomicNumber*, *atomicMass*, symbol, and *isMetal*. These parameters can be used to initialize the instance variables declared above. The *this* notation is used to reference the instance variables.

```
public Element(int atomicNumber, double atomicMass, String symbol, boolean isMetal){
        this.atomicNumber = atomicNumber;
        this.atomicMass = atomicMass;
        this.symbol = symbol;
        this.isMetal = isMetal;
}
```

Consider a class that can be used to model a DixieCup, or rather the contents that a DixieCup can hold. Your *DixieCup* class will need a *String* array to store the items in the cup. Recall, that because arrays in Java have a fixed length, we will also want a *boolean* type variable to keep track of whether our DixieCup is full (*isFull*).

Given the criteria above, our *DixieCup* class should have at least two instance variables: *String itemsArray[]*, and *boolean isFull*. These instance variables should be declared as private.

The *DixieCup* constructor should accept just one parameter. This parameter will represent the number of items the *DixieCup* can hold. In the body of the *DixieCup* constructor, you will use the number of items to initialize the *itemsArray[].* Each newly created *DixieCup* will be empty. This means that you will need to initialize *isFull* to false in your constructor.

Write the DixieCup class that meets the above requirements below.

□ Write an overloaded constructor

Recall that overloading in Java refers to the ability to create different objects from the same class depending on the types of parameters that are provided.

The constructor you wrote above is expecting a parameter which is then used to initialize the array of items the *DixieCup* can hold. But what if, we do not know how many items we want the DixieCup to hold? The user should have the ability to create empty DixieCups and then populate the DixieCup at a later time.

Write a second constructor below. This constructor should not accept any parameters. In the body of the constructor, *isFull* should be initialized to true and the number of items that can be stored in the itemsArray should be set to zero.

□ Write the *setIsFull* method

The job of the *setIsFull* method is to check whether or not a DixieCup is full. *setIsFull* does not need to return any information, so this method will be a "void" type method as shown below.

```
public void setIsFull(){
        //Write your code here
}
```

A *DixieCup* is full if there are no more null values in the *itemsArray*. If there are null values, *isFull* should be set to false, otherwise *isFull* should be set to true. Because we also have DixieCups that cannot hold anything, you will also need to check for this too.

□ Write the *addItem* method

Now that we have our constructors and check whether or not a cup is full, we can start adding contents to our Dixie cups.

We will do this by adding a method called *addItem*. The job of *addItem* is to simply add items to the cup. It does not need to return any information about the item, so this method will be a "void" type method as shown below.

```
public void addItem(String n){
        //Write your code here
}
```

Complete the *addItem* method. The *addItem* method should first check to see if the *DixieCup* can hold anything – that is, is *isFull* true or false. If is*Full* is false, you can add an item to your array. The item you add must go in the first *null* value of your *itemsArray*. You will need a loop to check for this. Once you have added your item, you should call *setIsFull*. *setIsFull* will set *isFull* to *true* if there are not anymore *null* places.

□ Write the *getItems* method

After we add items to our DixieCups, it would be nice to know the contents of our cups. To see the items in our cup we will create a *getItems* method. The job of the getItems method is simply to *return* the items in our DixieCup. Because the job of this method is to return something we need to specify that in the signature as shown below,

```
public String[] getItems(){
        //Write your code here
}
```

Write the *getItems* method which returns the *itemsArray.*

□ Write the *removeItem* method

Now that we know how to add items to the cups and see them. We want to have a method to remove them. Again, the job of *removeItem* is simply to remove an item, it does not return anything. Therefore, the signature for this method should look as follows,

```
public void removeItem(){
        //Write your code here
}
```

The *removeItem* method will remove the last item placed in the array. So, for an array that looks as follows, {paperclip, marble, eraser, null, null}, *removeItem* will remove the eraser by setting its position to null. For an array that is full, {paperclip, eraser, marble, penny}, the position of the penny would be set to null because it is the last item in the array. Be sure to call *setIsFull* after you remove an item to reset *isFull* as necessary.

□ Write the *DixieCupMaker* class

You can now use your *DixieCup* class to create Dixie cups full of items! We will do this with the *DixieCupMaker* class. The DixieCupMaker class will contain our main method. Because it is contains the main method, it is also referred to as the driver class.

```
public static void main(String args[]){
        //Write your code here
}
```

In the body of the *main* method we will create an array of dixie cups called dixieCupArray. The code below illustrates how to create an array of 5 DixieCup datatypes. DixieCup[] dixieCupArray = new DixieCup[5]; Once you have your array declared, write a loop to populate each index in your array with a new *DixieCup*. Recall, that some DixieCups will be able to hold items while others will not. Create a random number to determine whether the *DixieCup* should be able to hold something or not. The random number will determine which constructor you should use to create your *DixieCup*. For each *DixieCup* that can hold items, generate another random to determine how many items your cup can hold. Keep this value small – under 5 for example. Use this random number as the parameter for creating your *DixieCup* using the appropriate constructor.

□ Add items to your DixieCups

If your *addItem* method works correctly you should be able to add items to your cups.

Before you add items, you can also check to see what items are currently in your cup. To do this you can all the *getItems* method. But, recall that *getItems* returns an array. So, you will need to write a loop to examine the contents. The code below gets the items in the cup at index 2 and assigns them to the *cupItems* array. Once you have captured this array, you can write a loop to look at the contents.

String cupItems[] = dixieCupArray[2].getItems();

Use the *addItem* method to add a paper clip. Once you have done this, call the *getItems* method and write a loop to examine the contents and confirm that you have added the item.

□ Remove items from your DixieCups

To remove items from your cups, you will use the *removeItem* method. For example, to remove an item from the Dixie Cup I created above I could write,

```
dixieCupsArray[2].removeItem();
```

Write code that could be used to remove an item from the DixieCup at index 3. Then write code to display the contents of the DixieCup at index 3.

□ Receive credit for this lab guide

Submit this portion of the lab to Pluska to receive credit for the lab guide. Once received, your completed code challenges will also be graded and will count towards your final lab grade.