Problem 1: Functions

In this section just implement functions - do not worry about static or classes.

(1) A half-plane can be defined as the set of two-dimensional points (x, y) where $ax + by + c > 0$. Implement a function that takes in a half-plane defined by three floating point values $a, b, c$and a point determined by two floating point values x, y and returns if the point lies in the half-plane.

(2) Implement a function that returns the number of positive divisors a given integral number has (note that a is a divisor of b if $a\%b == 0$.

(3) A user wants to identify the number of times they have used '!' or '?' in a text they wrote. Write a function that uses switch/case to evaluate the number of those two characters appear in a String. Print the output out.

(4) Say we have two arrays of sorted integer numbers A and B. Construct a sorted array that contains all of the elements of A and B. This is called a merge, and is part of the merge sort algorithm.

Problem 2: Dataset

Lets implement a class that stores an array of data and can compute some statistics on that data.

```
interface AbstractDataset {

    // the number of values stored in the dataset
    public int size();

    // get the value stored at the corresponding index
    public double get(int index);

    // append a value to the dataset
    public double append (double value);

    // compute the average value
    public double getAverage();

    // compute the median value (use Collections.sort(List<T> list))
    public double getMedian();

    // compute the min and max value, returning an array of the form {min,max}
    public double[] getMinMax();

    // compute the range of values in the dataset (max - min)
    public double getRange();

}
```

(1) Declare a class called Dataset that implements AbstractDataset and will have a main function.

(2) Declare a member that can be easily resized that stores a bunch of double values.

(3) Specify a constructor that takes in no arguments.

(4) Specify a constructor that takes in a list of doubles.

(5) Finish implementing the interface

(6) Implement a main function that reads lines typed in by the user and outputs the above statistics after the user is done inputting data. The user is expected to only type in numbers or the word "exit". For example:

```
3.4
-2
3.45
20
16
exit
```

Problem 3: Matcher Factory

When parsing data typed in by a user it is quite common to have to identify bad inputs and return errors. We will call these tools "matchers". In this problem we will create an interface and factory to help developers easily construct matchers via a command-line-like interface. To do this we will create a MatcherFactory class that generates Matchers and then runs those matchers on a sequence of Strings through the matchers.

```
// simulates a user constructing two matchers
MatcherFactory factory = new MatcherFactory();
Matcher prefixMatcher = factory.create("prefix","User");
Matcher extensionMatcher = factory.create("extension", "txt");
Matcher[] matchers = new Matcher[]{prefixMatcher,extensionMatcher};

// simulates the user running these matchers through a series of inputs
String[] filenames = new String[]{"User2.txt","3.txt", "Username", "Usertxt"};

System.out.println("Factory has created " + factory.
for(String filename: filenames) {
    System.out.print(filename + ": ");
    for(Matcher matcher: matchers) {
        System.out.print("%b", matcher.match(filename));
    }
    System.out.println();
}
// With these inputs:
// > prefix User
// > extension txt
// > run User2.txt 3.txt Username Usertxt
// The code ouputs
// < 2 Filters
// < User2.txt: true true
// < 3.txt: false true
// < Username: true false
// < Usertxt: true false
```

(1) Declare a Matcher interface that requires a single function match. This function takes in a single string and returns whether whether the string passes the matcher or not.

(2) Implement a matcher called PrefixMatcher. This matcher makes sure that the input starts with the right string (called a prefix).

- The matcher stores a prefix which can be any sequence of characters.
- The constructor should take in the prefix.
- The match will return true if the string starts with the stored prefix. You do not need to match for capitalization.

```
// declare the class



// declare the members
```

```
// implement a constructor that populates the members


// implement necessary functions
```

(3) Implement a matcher called ExtensionMatch. This matcher matchers if the file has the right extension.

- This matcher stores an extension like "txt" or "doc" that does not contain any dots (".") at the end of the input.
- The constructor should take in the extension.
- The match will return true if the string contains a "." and the text after the "." matches the provided extension. You do not need to match for capitalization.

```
// declare the class


// declare the members


// implement a constructor that populates the members


// implement necessary functions
```

(4) Implement the MatcherFactory, which can create different types of matcher. The factory keeps track of the number of matchers it has created.

```
// declare the class




// Implement a member that counts the total number of matchers generated




// implement a getter for this counter member.









// implement a function for creating matchers using switch/case
// - "prefix" creates a PrefixMatcher where argument is the prefix
// - "extension" creates an ExtensionMatcher where the extension is argument
// don't forget ot increment the counter
Matcher create(String type, String argument) {









}
```

```
// Bonus: implement the main function that creates a dynamic list of matchers
// and runs those matchers on a set of inputs.
// It lines of text from System.in using a Scanner.
// Every line starts with a command
// - "prefix" adds a PrefixMatcher to the list of matchers
//              it takes one parameter (the prefix)
//              example: "prefix User"
// - "extension" adds an ExtensionMatcher to the list of matchers
//              it takes one parameter (the extension)
//              example: "extension doc"
// - "run" ,  print out the results of running the matchers against
//              all of the arguments.
//              it takes any number of parameters
//              example: "run User2.txt 3.txt Username Usertxt"
//
// Hint1: Use String.split(" ") to separate commands and arguments
// Hint2: Use an if/else to determine the commands
// Hint3: Look at the example code for this problem for advice.
```

Problem 4: Cipher

A simple form of cryptography is to transform each value according to a function that we know how to invert. Your job is implement a class that uses a Cipher and then implement a Cipher yourself. Lets say we have an interface called Cipher defined by

```
public interface CharacterCipher {
    // encodes a int according to some algorithm
    public char encode(char c);
    // encodes an char according to some algorithm such that
    // c == decode(encode(c))
    public char decode(char c);
}
```

(1) Please write a class that can encode or decode an array of numbers called VectorCipher. Its constructor takes in a Cipher and it uses this Cipher to implement its encode or decode. Note that if the character is a space then that character should not be encoded.

Implement this by following most of the VectorCipher interface. Use a for loop for encode and a while loop for decode.

```
public interface VectorCipherInterface {
    // encodes every number int the input array using a specific cipher
    public int[] encode(int[] input);
    // decode an encoded int array using a specific cipher
    public int[] decode(int[] encoded);
}

// declare a class called VectorCipher




// Specify the member cipher



// Specify a constructor that takes in the member



// Implement encode using a for loop









// Implement decode using a while loop
```

(2) The Caesar cipher is one of the simplest forms of cryptography and was apparently used by Julius Caesar in his correspondences. A cipher is defined by an"offset" which specifies how many characters to "shift" an input string by. For integers this comes down to:

$$encode(d) = d + offset \tag{1}$$
$$decode(d) = d - offset. \tag{2}$$

```
// Define a Cipher called CaesarCipher


// Implement a member for the offset


// Implement a constructor for the CaesarCipher that takes in the offset




// Implement the rest of the interface
```

(3) Implement a function called encodeWord that encodes a word using the Caesar cipher with a given offset

```
String encodeWord(String str, int offset) {
    // create a CaesarCipher


    // create an array of characters to store the encoded result


    // encode each character in the string


    // return a new String based on the character array

}
```