Assignment 14
Programs 14-1, 14-2 and 14-3 are due May 6
Program 14-4 is Due May 9

Program 14-1:

There are three components to this problem

1. **A method to convert infix expressions to postfix expressions**
2. **A method to evaluate postfix expressions**
3. **A Gui**

Part 1 Start with a method that converts infix to postfix.
The algorithm is in the notes

Make a class ConvertToPostfix with one static method

```
public static String  convert(String infix)
```

that returns the postfix version of an infix string.

For example, if infix is "20+30*40" then
    convert(infix) returns " 20 30 40 * + "

As in the examples from class, we separate the infix string into tokens.
If the infix string is "20+30*40"
The tokens are "20"   "+"   "30"   "*"   and "40 "

Java has a class – StringTokenizer-- that makes getting the tokens easy.

A StringTokenizer object separates a string into "tokens" or components --- as long as you tell the StringTokenizer object **what separates the tokens**. For example, if the "separator" is a space then the tokens of the string "Abba Dabba Dabba" are

Abba
Dabba
Dabba

If the separators are + - * / (and) then the tokens of (345+567)*5 are
345
567
5

The list of separators is called the *delimiters*. *There can be any number of delimiters*

For this problem you will use a StringTokenizer (in java.util) object to break an infix expression into its tokens, where

- the tokens are **both** numbers as well as + - * /) (.
- The delimiters or separators are + - * /) and (.

For example the tokens of (456+ 67)* 3 are
(
456
+
67
)
*
3

Notice that the operators and paren are both tokens and separators.

To create a StringTokenize object use the constructor:

```
StringTokenizer st =
                new StringTokenizer(String expression,  ")(+-*/",  true);
```

- The string ")(+-*/" (in the constructor) lists the separators,
- the value **true** means that the separators are also tokens..

So, the expression (345+23)*45 has as tokens

(
345
+
23
)
*
45

There are two methods that are important

- boolean hasMoreTokens()
- String nextToken()

Here is an example that takes an expression and prints the tokens

```
String expression = "345  +   99*(36 +  2 );
StringTokenizer  st =    new StringTokenizer(expression,   ")(*+-/",   true);

while (st.hasMoreTokens())
{
        String token = st.nextToken();
        token = token.trim();           // strips off any whitespace;
        System.out.println(token);

}
```

Output is:

```
345
+
99
*
(
36
+
2
)
```

Notice the trim method that strips away unnecessary whitespace from a token.
Here is the format of your program. You can include other methods if you like.

```
public class ConvertToPostfix
{
   public static String convert(String infix)
   {
         Stack<String> s = new Stack<String>(); // or Stack<String>(100) if you use the array

         StringTokenizer st = new StringTokenizer(infix,  ")(+-*/",  true);
         String postfix = "";
          while (st.hasMoreTokens())
          {
                 String token = st.nextToken();
                 token = token.trim();
                 //  your code goes here
          }
         return postfix;
     }
}
```

Here is a separate class to test your method. Remember that you call a static method with the class name

```
public class TestConvert
{
          public static void main(String[] args)
      {
          Scanner input = new Scanner(System.in)
          System.out.print("Enter infix  -- end with XXX");
           String infix = input.nextLine();
           while (!infix.equals("XXX")
           {
                      System.out.println("Postfix is  "+ ConvertToPostfix.convert(infix);
                      System.out.print("Enter infix  -- end with XXX");
                     infix = input.nextLine();

           }
}
```

Don't go any further until you are convinced your convert() method is correct.

Part 2. Once you have a convert() method working, it is time to write a method to evaluate the postfix expression.

Make a class Postfix with a single static method

```
public static int evaluate (String postfix)
```

that accepts a postfix expression and returns its numerical value.

Remember that components/tokens of the postfix expression are separated by spaces. For example,

```
345 71 + 10 *
```

Here you will use a Scanner to read from the postfix string. (You did this once before)

Suppose that the postfix expression is

```
String postfix = " 10 20 * 50 +"
```

If you instantiate the Scanner with a String (and not System.in or a file), it will read from the String instead of from System.in or a file.

For example, assume postfix = " 10 20 * 50 +"

```
Scanner input = new Scanner(postfix);     // not System.in
while (input,hasNext())
{
        String s = input.next();  // reads from postfix string
        System.out.println(s);
}
```

prints

```
10
20
*
50
 +
```

In other words, input.next() skips whitespace and returns each individual component of the postfix string.

Here is an outline of the class with a main method for testing

```
 public class Postfix
{

        public static int evaluate( String postfix)
        {
                 Stack<Integer> s = new Stack<Integer>(100); // or Stack<Integer>()
                //or Stack<Integer>() if using linked implementation

              // your code

        }

        public static void main(String[] args)  // for testing
        {

                 String s = "20 10 + 30 + 2 *";
                 System.out.println(s + " = " + Postfix.evaluate(s));
```

```
            s = "30 40 + 50 - 10 +";
            System.out.println(s + " = " + Postfix.evaluate(s));

            s = " 20 10 3 * + 50 + 2 *";
            System.out.println(s + " = " + Postfix.evaluate(s));

            s = " 5 2 / 6 + 10 -";
            System.out.println(s + " = " + Postfix.evaluate(s));

           s = " 50 10 * 4 / 2 * 7 3 - +";
            System.out.println(s + " = " + Postfix.evaluate(s));

    }
}
```

Note: When using this method in another class, because it is static, you call it with the class name → Postfix.evaluate(…)

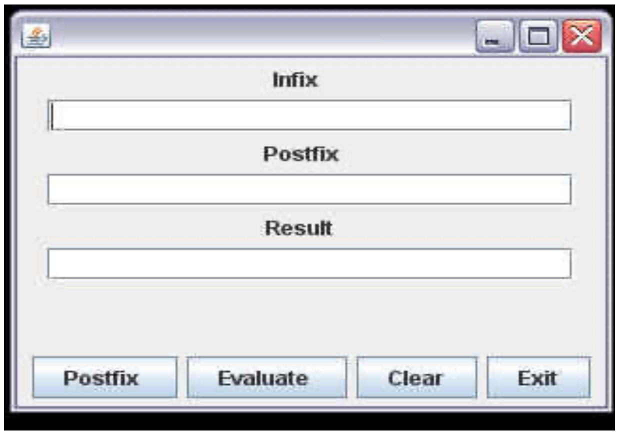
Part 3 The GUI

You have written two static methods

1. String ConvertToPostfix.convert (String infix) // returns postfix
2. Int Postfix.evaluate(String postfix) // returns value of postfix

Now we will use both of these in one program:

Your program should display a fixed-size (not resizable) frame with four buttons and three text fields:



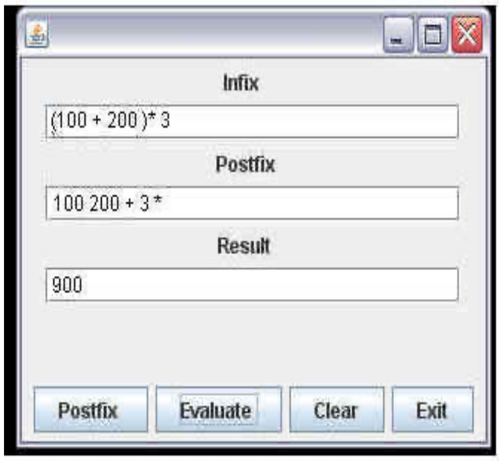
The **Postfix button** converts the infix expression (in the Infix text field) to postfix and displays the postfix expression into the postfix field.

The **Evaluate button** takes the postfix expression, evaluates it and places the result in the Result text field.

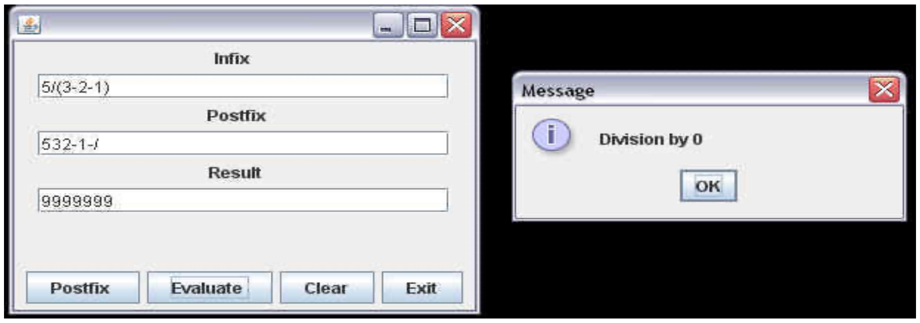
The **Clear button** sets each text field to the empty string.

If a text field is empty and a button pressed, no action should occur.

Here is a picture after an infix string is placed in the infix text field and the Postfix and Evaluate buttons pressed:


Infix
(100 + 200)* 3
Postfix
100 200 + 3 *
Result
900

1. Assume an infix string is correct but you may still run into division by 0.
In that case, use JOptionPane.ShowMessage(...) and place 9999999 in the result field


Infix
5/(3-2-1)
Postfix
532-1-/
Result
9999999
Message
Division by 0

Notice that we wrote the methods to convert and evaluate separately and not in the listener classes. It is good programming practice to separate the logic of a program from the G

Problem 14- 2: Deques using a circular array

Implement a deque using a circular array. This is very similar to the queue implementation. If you draw pictures and see how things work the coding will be easier.

You just have to add three new methods to the queue class the

- **void insertFront(E x)**
- **E removeRear()**
- **E peekRear();**

insertRear(E x) is the same as insert(E x) for a queue
E removeFront () is the same as E remove() for a queue
E peekFront() is the same as E Peek() for a queue

Here is most of the class. Most of it is **exactly** the same as the Queue class   You just add the three new "red methods."

```
public class Deque<E>
{
   private E[] items;
   private int dequeSize;
   int front, rear;
   int max;

   public Deque()      // default constructor, same as a queue
   {
      items =(E[]) new Object[10];
      dequeSize= 0;
      front = rear= -1;
      max = 10;
   }

   public Deque(int max)     // one argument constructor, same as queue
   {
      this.max = max;
      items =(E[]) new Object[maxQueue];
      deque = 0;
      front = rear= -1;
   }

   public void insertRear(E x)  // same as insert(..) for queue
   {
      if ( dequeSize == max) // queue is full
      {
         System.out.println(" Queue Overflow");
         System.exit(0);
```

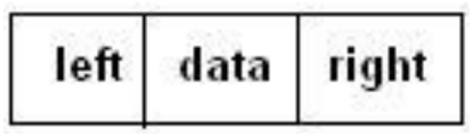
```
    }

    rear = (rear +1) % max;
    items[rear] = x;
    dequeSize++;
    if (dequeSize == 1)
        front = rear;
}

  public E removeFront() // same as remove() for queue
{
    if (dequeSize == 0)
      {
        System.out.println(" Queue Underflow");
        System.exit(0);
       }

    E temp = items[front];
    dequeSize--;
    if ( dequeSize == 0)
        front = rear = -1;
    else
        front = (front + 1) % max;
    return temp;
}

public E peekFront() // same as peek() for queue

{
    if (dequeSize == 0)
    {
        System.out.println(" Queue Underflow");
        System.exit(0);
    }
        return items[front];
}

public boolean empty()

{
    return dequeSize == 0;
}

public int size()

{
        return dequeSize;
    }

   public void insertFront(E x)
   {
        // your code
   }

  public E removeRear()
  {
        // your code
   }
   Public peekRear()
   {
        // your code
   }
}
```

Test the Deque class by adding the following main() method to the class:

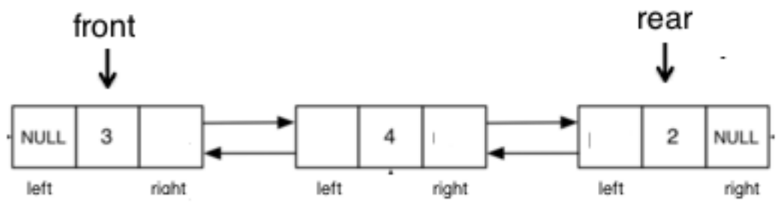
```
public static void main(String[] args)
  {
    Deque<Integer> q = new Deque<Integer>();
    for(int i = 0; i <=5; i++)
      q.insertFront(i);

    for(int j = 10; j <= 15; j++)
      q.insertRear(j);

    for(int k = 1; k <= 5; k++)
    {
      System.out.println(q.removeFront());
      System.out.println(q.removeRear());
    }

    System.out.println("size is now  "+  q.size());

    while(!q.empty())
      System.out.println (q.removeRear());

    System.out.println("size is now  "+  q.size());
  }
```

Problem 14-3 Deques using a “doubly linked list”

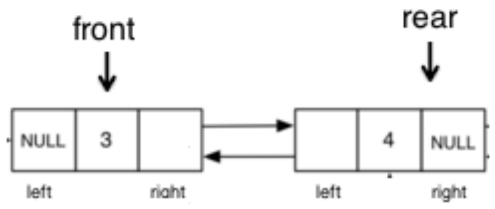
Implement the deque as chain of nodes.
As we saw in class, unlike an ordinary queue a Node will have two reference fields :
left and right (as well as a data field).



The ***left*** field points to a node to the left (or is null); the ***right*** field points to a node to the right (or is null).



Notice that the removeRear() operation leaves the deque as



So moving the rear pointer to the left is precisely why we use nodes with two reference fields.

Here is a skeleton of the class…**you fill the details.**
If you draw pictures, you will have a better understanding of how this works.
Looking at the linked implementation of a queue will also help. They are similar
Call the class Deque1.java to differentiate it from the previous implementation

```
public class Deque1<E>
{
  private class Node  // has 2 reference fields , left and right
 {
    private E data;
    private Node left;
    private Node right;

    public Node()  // default constructor
    {
        // your code
    }
    public Node (E x) // one argument constructor for Node
    {
        // your code
     }
  }

  Node  front, rear;  // global variables
  int dequeSize;

  public Deque() // default constructor
  {
    front = rear = null;
    dequeSize = 0;
  }

  public void insertFront(E x)
  {
      // Create a Node
      if (dequeSize == 0)             //  check special case
       // your code goes here; look at the queue, code is similar
      else  // not initially empty
      {
        // your code goes here; three lines
        // remember that a node has 2 reference fields.
       }
       // adjust the size of the deque
   }
 public void insertRear(E x)
{
   // create a Node
   if (dequeSize == 0)                // check special case
      // your code goes here
     else                             // not initially empty
     {
       // your code goes here, similar to insertFront(..)
      }
      // adjust the size of the deque
 }
}
public E removeFront()
{
  // if empty, cannot remove—write code for that
  // store value to be returned in a temp
  // move front to the right  and  then set front.left  to null
  // adjust dequeSize
  // if the dequeSize is now 0 ,  the deque is empty, adjust front and rear
  // return temp
 }

 public E removeRear()
 {

    // very similar and symmetrical to removeFront()

 }

 public boolean empty()
 {
   // one line
 }

   public int dequeSize()
   {
        // one line
   }
  // Add this main method for testing
public static void main(String[] args)
  {
    Deque<Integer> q = new Deque<Integer>();
    for(int i = 0; i <=5; i++)
      q.insertFront(i);
    for(int j = 10; j <= 15; j++)
      q.insertRear(j);
    for(int k = 1; k <= 5; k++)
    {
      System.out.println(q.removeFront());
      System.out.println(q.removeRear());
    }
    System.out.println("dequeSize is now  "+  q.dequeSize());

    while(!q.empty())
      System.out.println (q.removeRear());

    System.out.println("dequeSize is now  "+  q.dequeSize());
  }
}
```

Problem 14-4 : Simulation

Here you will extend the ATM simulation.

You can get Queue at the OtherSuff link on the class website.
The Customer.java and ATMSimulation.java also on that site.

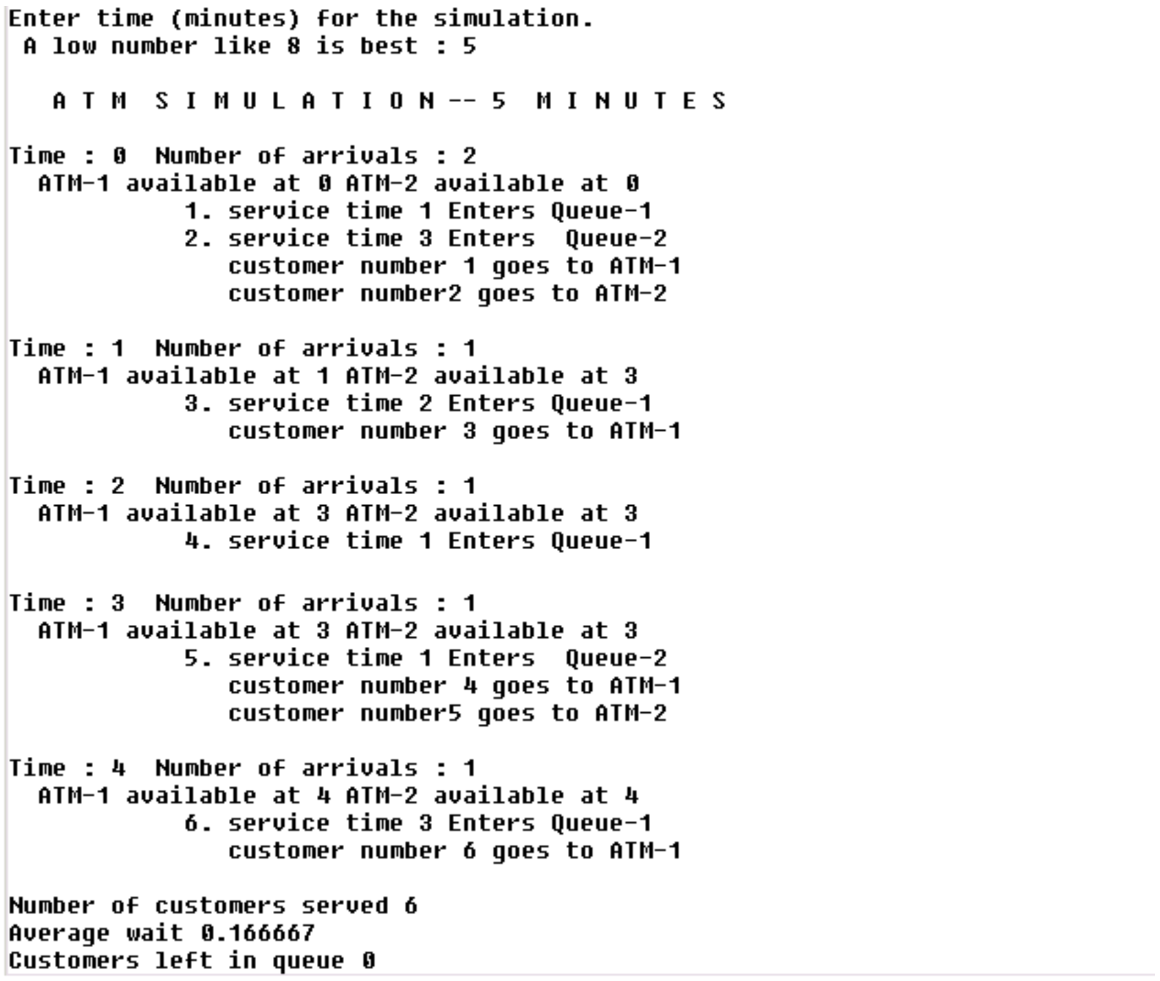
Just scroll down to “Programs from Class.”

Here is your assignment:

Assume that the bank has installed a second ATM machine. **Each ATM machine has its own waiting line.** When a customer enters the system, he/she picks the shorter line. (If both lines are the same size, pick the first.) All the other assumptions (number of customers, service times etc.) are the same.

Your program should print out the actions for each minute. Do NOT simulate for an hour. A low number, such 5 - 8 is enough.

Here is a screenshot of the simulation. Your output should display the same information, but not necessarily the same data.



```
Enter time (minutes) for the simulation.
 A low number like 8 is best : 5

   A T M  S I M U L A T I O N -- 5  M I N U T E S

Time : 0  Number of arrivals : 2
  ATM-1 available at 0 ATM-2 available at 0
            1. service time 1 Enters Queue-1
            2. service time 3 Enters  Queue-2
               customer number 1 goes to ATM-1
               customer number2 goes to ATM-2

Time : 1  Number of arrivals : 1
  ATM-1 available at 1 ATM-2 available at 3
            3. service time 2 Enters Queue-1
               customer number 3 goes to ATM-1

Time : 2  Number of arrivals : 1
  ATM-1 available at 3 ATM-2 available at 3
            4. service time 1 Enters Queue-1

Time : 3  Number of arrivals : 1
  ATM-1 available at 3 ATM-2 available at 3
            5. service time 1 Enters  Queue-2
               customer number 4 goes to ATM-1
               customer number5 goes to ATM-2

Time : 4  Number of arrivals : 1
  ATM-1 available at 4 ATM-2 available at 4
            6. service time 3 Enters Queue-1
               customer number 6 goes to ATM-1

Number of customers served 6
Average wait 0.166667
Customers left in queue 0
```