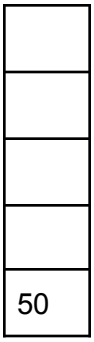
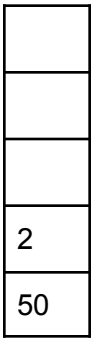
Question 1

a) Consider the following Stack and draw the Stack frames after executing each statement given below.

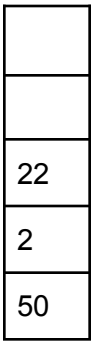
int a = 22, b = 44;



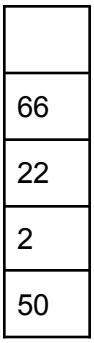
i) theStack.push(2);



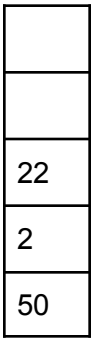
ii) theStack.push(a);



iii) theStack.push(a + b);

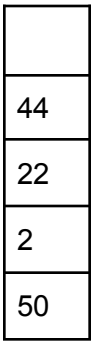


iv) theStack.pop();



Return 66

v) theStack.push(b);



vi) theStack.push(a –b);

50

Question 2

i) A stack class has already been implemented with push() , pop() and peek() methods. It is used to store characters. Write a code segment in your main application to insert following characters to a 'myStack' object created from the stack class. 'g' , 't', 'o', 'p'

```
myStack.push('g');
myStack.push('t');
myStack.push('o');
myStack.push('p');

while(!myStack.isempty()){
        sout(myStack.pop());
}
```

ii) Write code segment to display all the values in a stack by removing them.

iii) What is the result of section ii) above?

p
o
t
g

Question 3

A stack class called StackX has been created to store characters. 'push' and 'pop' methods have been implemented. Implement the peek method of StackX class using push and pop methods.

```
Public char peek(){
        if(top == -1){
                Return 'err';
        }
        Else{
                Char temp = this.pop();
                this.push(temp);
                Return temp;
        }
}
```

Additional Exercises:

Question 1

i) Implement a class called StackX to store a set of characters.

```
// reverse.java
// stack used to reverse a string
// to run this program: C>java ReverseApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class StackX
 {
 private int maxSize;
 private char[] stackArray;
 private int top;
//--------------------------------------------------------------
-
 public StackX(int max) // constructor
 {
 maxSize = max;
 stackArray = new char[maxSize];
 top = -1;
 }
//--------------------------------------------------------------
-
 public void push(char j) // put item on top of stack
 {
 stackArray[++top] = j;
 }
//--------------------------------------------------------------
-
 public char pop() // take item from top of stack
 {
 return stackArray[top--];
 }
//--------------------------------------------------------------
-
```

```
 public char peek() // peek at top of stack

 {
 return stackArray[top];
 }
//--------------------------------------------------------------
-
 public boolean isEmpty() // true if stack is empty
 {
 return (top == -1);
 }
//--------------------------------------------------------------
-
```

ii) Create a class called Reverser to reverse a given string using the stack class created above.

class Reverser {

private String input; private String output; ……..

}

(Hint: Pass the string to be reversed as an argument to the constructor and store it in input)

```
class Reverser
 {
 private String input; // input string
 private String output; // output string
//--------------------------------------------------------------
-
 public Reverser(String in) // constructor
{
      input = in;
}
//--------------------------------------------------------------
-
 public String doRev() // reverse the string
 {
```

```
 int stackSize = input.length(); // get max stack size
 StackX theStack = new StackX(stackSize); // make stack
 for(int j=0; j<input.length(); j++)
 {
 char ch = input.charAt(j); // get a char from input
 theStack.push(ch); // push it
 }
 output = "";
 while( !theStack.isEmpty() )
 {
 char ch = theStack.pop(); // pop a char,
 output = output + ch; // append to output
 }
 return output;
 } // end doRev()
//--------------------------------------------------------------
-
 } // end class Reverser
```

iii) In main() get a string from the user and reverse the string using the Reverser class.

```
 class ReverseApp
 {
 public static void main(String[] args) throws IOException
 {
 String input, output;
 while(true)
 {
 System.out.print("Enter a string: ");
 System.out.flush();
 input = getString(); // read a string from
kbd
 if( input.equals("") ) // quit if [Enter]
 break;
 // make a Reverser
 Reverser theReverser = new Reverser(input);
 output = theReverser.doRev(); // use it
 System.out.println("Reversed: " + output);
```

```
 } // end while
 } // end main()
//--------------------------------------------------------------
-
 public static String getString() throws IOException
{
 InputStreamReader isr = new InputStreamReader(System.in);
 BufferedReader br = new BufferedReader(isr);
 String s = br.readLine();
 return s;
 }
//--------------------------------------------------------------
-
 } // end class ReverseApp
```

Question 2
Use the stack class created in Question1 (i) and check whether a user entered expression is correctly parenthesized.

Ex: 3 + ((6 * 2) – 3) valid
5 * 6 + (2 - 5 not valid

```

 class BracketChecker
 {
 private String input; // input string
//--------------------------------------------------------------
-
 public BracketChecker(String in) // constructor
 { input = in; }
//--------------------------------------------------------------
-
 public void check()
 {
 int stackSize = input.length(); // get max stack size
 StackX theStack = new StackX(stackSize); // make stack
```

```
 for(int j=0; j<input.length(); j++) // get chars in turn
 {
 char ch = input.charAt(j); // get char
 switch(ch)
 {
Note that here , all the case statements are executed from the first until
the break
 case '{': // opening symbols
 case '[':
 case '(':
 theStack.push(ch); // push them
 break;
 case '}': // closing symbols
 case ']':
 case ')':
 if( !theStack.isEmpty() ) // if stack not
empty,
 {
 char chx = theStack.pop(); // pop and check
 if( (ch=='}' && chx!='{') ||
 (ch==']' && chx!='[') ||
 (ch==')' && chx!='(') )
 System.out.println("Error: "+ch+" at "+j);
 }
 else // prematurely empty
 System.out.println("Error: "+ch+" at "+j);
 break;
 default: // no action on other characters
 break;
 } // end switch
 } // end for
 // at this point, all characters have been processed
 if( !theStack.isEmpty() )
 System.out.println("Error: missing right delimiter");
 } // end check()
//--------------------------------------------------------------
-
```

```
} // end class BracketChecker
```

Stack Example 2: Delimiter Matching

One common use for stacks is to parse certain kinds of text strings. Typically the strings are lines of code in a computer language, and the programs parsing them are compilers.

To give the flavor of what's involved, we'll show a program that checks the delimiters in a line of text typed by the user. This text doesn't need to be a line of real Java code

(although it could be) but it should use delimiters the same way Java does. The delimiters are the braces '{'and'}', brackets '['and']', and parentheses '('and')'. Each opening or left delimiter should be matched by a closing or right delimiter; that is, every '{' should be followed by a matching '}' and so on. Also, opening delimiters that occur later in the string should be closed before those occurring earlier. Examples:

```
c[d]        // correct
a{b[c]d}e   // correct
a{b(c]d}e   // not correct; ] doesn't match (
a{b{c}d]e}  // not correct; nothing matches final }
a{b(c)      // not correct; Nothing matches opening {
```

Opening Delimiters on the Stack

The program works by reading characters from the string one at a time and placing opening delimiters, when it finds them, on a stack. When it reads a closing delimiter from the input, it pops the opening delimiter from the top of the stack and attempts to match it with the closing delimiter. If they're not the same type (there's an opening brace but a closing parenthesis, for example), then an error has occurred. Also, if there is no opening delimiter on the stack to match a closing one, or if a delimiter has not been matched, an error has occurred. A delimiter that hasn't been matched is discovered because it remains on the stack after all the characters in the string have been read.

Let's see what happens on the stack for a typical correct string:

```
a{b(c[d]e)f}
```

Table 4.1 shows how the stack looks as each character is read from this string. The stack contents are shown in the second column. The entries in this column show the stack contents, reading from the bottom of the stack on the left to the top on the right.

As it's read, each opening delimiter is placed on the stack. Each closing delimiter read from the input is matched with the opening delimiter popped from the top of the stack. If they form a pair, all is well. Nondelimiter characters are not inserted on the stack; they're ignored.

Table 4.1: Stack contents in delimiter matching

Character Read	Stack Contents
A	
{	{
B	{
(	{(
C	{(
[	{([
D	{([
]	{(
E	{(
)	{
F	{
}	