Inversor de Cadenas

```
public class Inversor {

    private String entrada;
    private String salida;

    public Inversor(String entrada) {
        this.entrada = entrada;
    }
    public String invertir()
    {
        Pila pila = new PilaVec();
        char ch;

        for (int j = 0; j < entrada.length(); j++) {
            ch = entrada.charAt(j);
            Character caracter = new Character(ch);
            pila.apilar(caracter);
        }
        salida = "";
        while (!pila.esVacia()) {
            try {
             Character tmp = (Character) pila.cimaYDesapilar();
             salida += tmp;
            } catch (DesbordamientoInferior e) {
                e.printStackTrace();
            }
        }
        return salida;
    }
}
```

==========

```
public class InversorCadenas {

    public static void main(String[] args) {

        String entrada;
        String salida;

        entrada = "abcdefg";
        Inversor inversor = new Inversor(entrada);
            salida = inversor.invertir(); // use it
            System.out.println("Original: " + entrada);
            System.out.println("Invertida: " + salida);
    }
}
```

==========

```
Original: abcdefg
Invertida: gfedcba
```

Palíndromo

```
/*Palíndromo.
Palabra o frase que se lee igual de izquierda a derecha,
que de derecha a izquierda;*/

public class Palindromo {

    public static void main(String[] args) {

        char ch;
        String entrada = "AB ba";        // entrada
        int numeroLetras = 0;
        Pila pila = new PilaVec();  //guarda caracteres no blancos
        Cola cola = new ColaVec();  //guarda caracteres no blancos

        for (int i = 0; i < entrada.length(); i++) {
            ch = entrada.charAt(i);
            if (Character.isLetter(ch)) {
                numeroLetras++;
                ch = Character.toLowerCase(ch);
                pila.apilar(new Character(ch));
                cola.insertar(new Character(ch));
            }
        }
        boolean esPalindromo = true;
        int numChars = 0;

        while (esPalindromo && (numChars < numeroLetras)) {
            try {
                Character deLaPila = (Character) pila.cimaYDesapilar();
                Character deLaCola = (Character) cola.quitarPrimero();
                if (!deLaPila.equals(deLaCola))
                    esPalindromo = false;
                numChars++;
            } catch (DesbordamientoInferior e) {
                e.printStackTrace();
            }
        }
        if (!esPalindromo) {
            System.out.println(entrada+" --->No es palíndromo ");
        } else {
            System.out.println(entrada+" --->Es palíndromo ");
        }
    }
}
```

```
AB ba --->Es palíndromo
```

Símbolos balanceados

```
public class SimbolosBalanceados {
    public static boolean caracterApertura(char ch){
        return ((ch == '(') || (ch == '[') || (ch == '{'));
    }
    public static boolean caracterCierre(char ch){
        return ((ch == ')') || (ch == ']') || (ch == '}'));
    }
    public static void main(String[] args) {
        boolean stringBalanceado = true;
        char simboloApertura;
        int indiceCharActual = 0;
        char charActual;

        Pila pila = new PilaVec();
        String linea = "((((([]aa)))";
        int ultimoIndiceChar = linea.length() - 1;
        while (stringBalanceado && (indiceCharActual <= ultimoIndiceChar)) {
            charActual = linea.charAt(indiceCharActual);
            System.out.print(charActual);
            if (caracterApertura(charActual)){
                // wrap el caracter y se mete en la pila
                Character simbolo = new Character(charActual);
                pila.apilar(simbolo);
            } else {
                if (caracterCierre(charActual)){
                    try {
                        simboloApertura = ((Character) pila.cimaYDesapilar()).charValue();

                        if (!(((charActual == ')') && (simboloApertura == '('))
                                || ((charActual == ']') && (simboloApertura == '['))
                                || ((charActual == '}') && (simboloApertura == '('))))
                            stringBalanceado = false;
                    } catch (Exception e) {
                        stringBalanceado = false; // pila vacia
                    }
                }
            }
            indiceCharActual++; // siguiente caracter
        }
        if (!stringBalanceado) {
            System.out.println(" Simbolos no balanceados ");
        } else if (!pila.esVacia()) {
            System.out.println(" Final prematuro String");
        } else {
            System.out.println(" String  balanceado");
        }
    }
}
```

```
((((([]aa))) Final prematuro String
```

```
((([]aa))) String  balanceado
```

Calculadora Sencilla

- ✓ **Requisito:** La calculadora debe implementar **expresiones infijas.**
- ✓ **Requisito**: Evalúa enteros y hace las operaciones de suma, resta, multiplicación, división y potencia.
- ✓ **Requisito**: Las operaciones tienen **precedencia** entre ellas. Además se pueden manejar **paréntesis** como operador de máxima precedencia.

Expresiones infijas.

Son expresiones de tipo 1 + 2 * 3

que se evalúa a 7 y no a 9 porque la operación de multiplicación tiene mayor precedencia que la suma.

10 – 4 – 3 que se evalúa en 3. Precedencia de izquierda a derecha.

La potencia se evalúa de derecha a izquierda:

2^3^3 seria 2 ^ (3^3) en lugar de (2^3)^3

Dificultad para evaluar expresiones infijas:

1 – 2 – 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2

que es equivalente a

(1 – 2) – (((4 ^ 5) * 3)* 6) / (7 ^ (2 ^ 2)))

Que se evalúa a 8

Expresiones postfijas.

Están formadas por una serie de operadores y operandos, que evalúan directamente en una **máquina postfija**

Ejemplo:

1 2 3 * + se evalúa a 7

Funcionamiento de una máquina postfija.

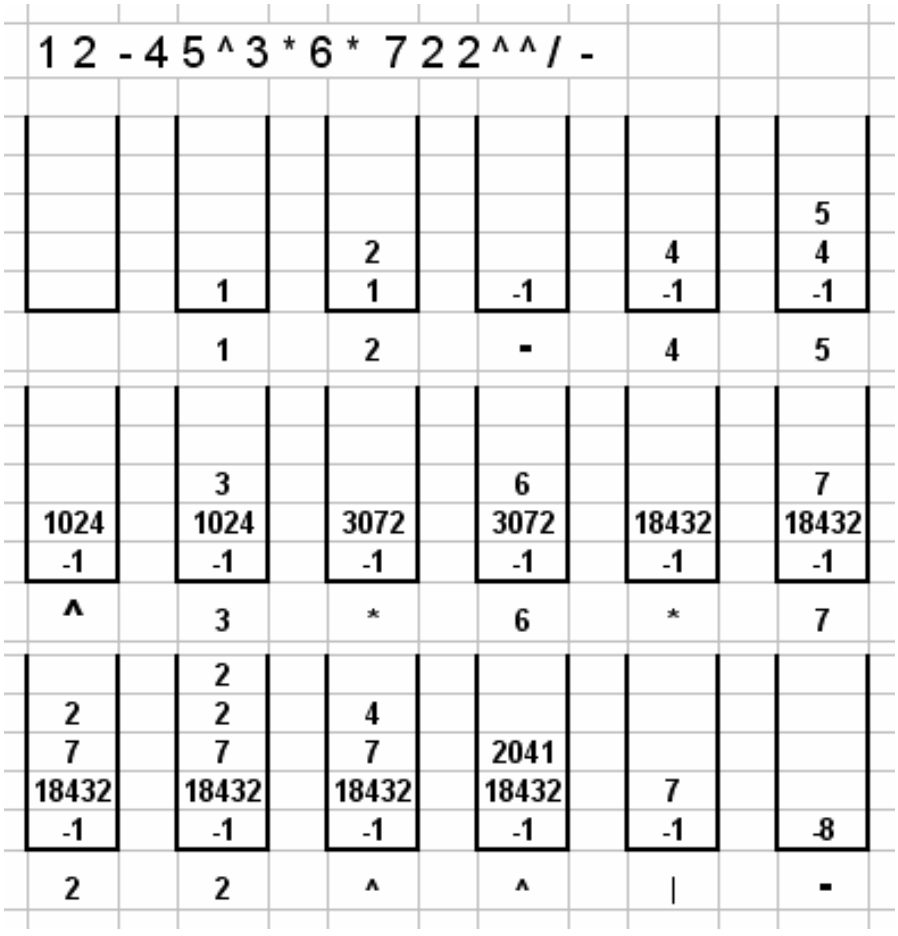
1. Los operandos se van apilando en una pila.
2. Al encontrar un operador, se desafilan sus operandos y se apila el resultado.
3. Al finalizar la evaluación, en la pila quedara un valor que es el resultado.

Ejemplo:

Expresión infija 1 – 2 – 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2

(1 – 2) – (((4 ^ 5) * 3)* 6) / (7 ^ (2 ^ 2)))

Expresión postfija 1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -



9 operandos, 8 operadores =➔ 17 pasos con 17 apilamientos.

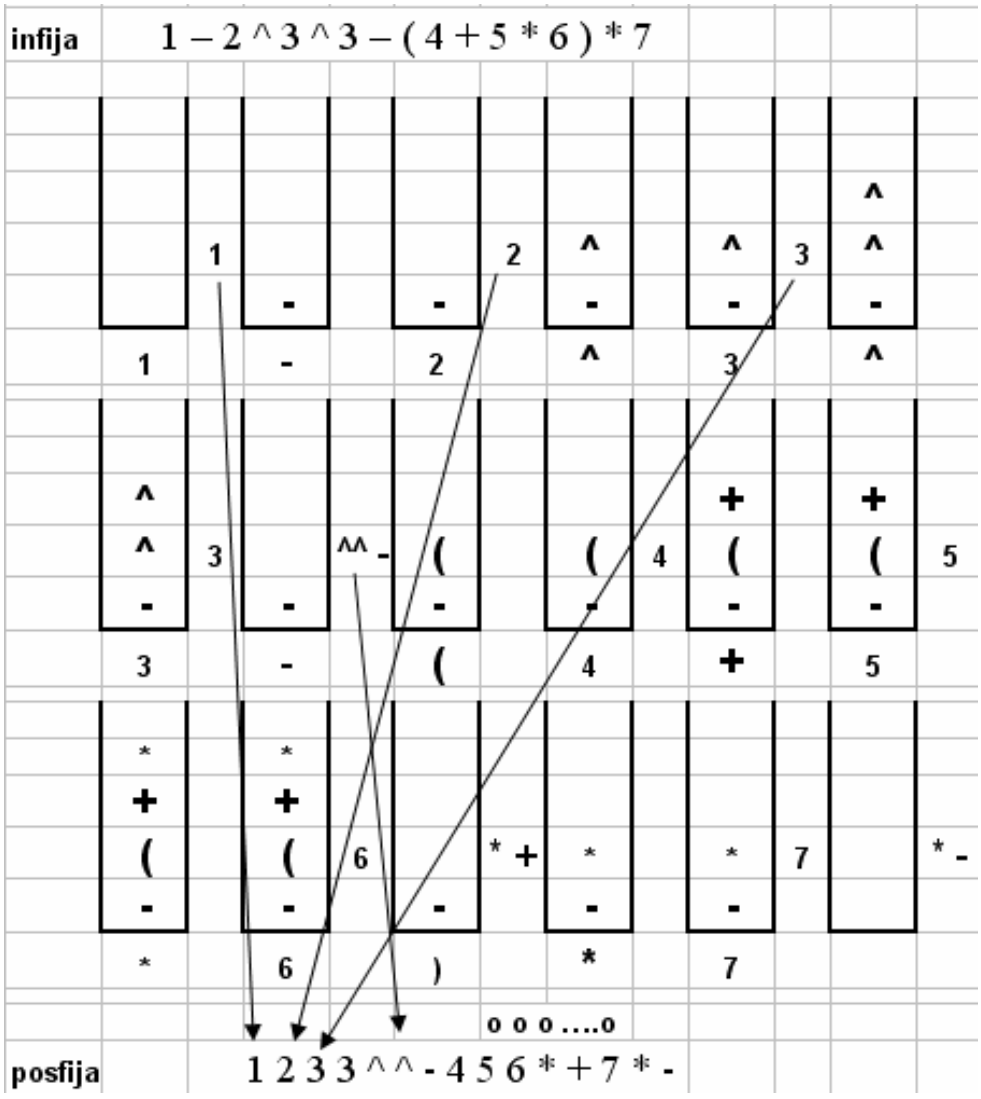
El tiempo necesario para evaluar una expresión postfija es **lineal. O(N)**

Conversión de notación infija a postfija

Resumen del algoritmo de análisis sintáctico de expresiones con precedencia de operadores:

1. **Operandos** : Pasan inmediatamente a la salida.
2. **Paréntesis derecho**: Desapilar símbolos hasta encontrar un paréntesis izquierdo.
3. **Operador:** Desapilar todos los símbolos hasta que encontremos un símbolo de menor precedencia o un símbolo de igual precedencia con asociatividad por la derecha. Apilar entonces el operador encontrado.
4. **Fin de la entrada**: Desapilar el resto de los símbolos en la pila.

Ejemplo: 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7



La clase precedencia

```
class Precedencia {

    int simboloEntrada;

    int cimaPila;
    /**
     * Constructor
     * @param simEnt precedencia para entrada
     * @param simCima precedencia en pila
     */
    Precedencia(int simEnt, int simCima) {
        simboloEntrada = simEnt;
        cimaPila = simCima;
    }
}
```

Clase Evaluador

```
public class Evaluador {

    /** Final de linea */
    static final int EOL = 0;

    /** Valor numérico */
    static final int VALOR = 1;

    /** Parentesis de apertura */
    static final int PAREN_A = 2;

    /**Parentesis de cierre */
    static final int PAREN_C = 3;

    /**Potencia*/
    static final int EXP = 4;

    /** Multiplicación */
    static final int MULT = 5;

    /** Division */
    static final int DIV = 6;

    /** Suma */
    static final int MAS = 7;

    /** Resta */
    static final int MENOS = 8;
    //tablaPrec genera el orden de evaluación de los simbolos
    static Precedencia[] tablaPrec = new Precedencia[9];
    static {
        tablaPrec[0] = new Precedencia(0, -1);   //EOL
        tablaPrec[1] = new Precedencia(0, 0);    //VALOR
        tablaPrec[2] = new Precedencia(100, 0);  //PAREN_A
        tablaPrec[3] = new Precedencia(0, 99);   //PAREN_C
        tablaPrec[4] = new Precedencia(6, 5);    //EXP
        tablaPrec[5] = new Precedencia(3, 4);    //MULT
        tablaPrec[6] = new Precedencia(3, 4);    //DIV
        tablaPrec[7] = new Precedencia(1, 2);    //MAS
        tablaPrec[8] = new Precedencia(1, 2);    //MENOS
    }
    private Pila pilaOperador; //Pila de operadores para la conversión
    private Pila pilaPostfija; //Pila para la maquina postfija
    StringTokenizer entrada;   //La cadena de caracteres
    long valorActual;          //Operando actual
    int ultimoToken;           //Último simbolo leído
```

```java
/**
 * Construye un objeto evaluador
 * @param expresion la cadena que contiene la expresión
 */
public Evaluador(String expresion) {
    this.pilaOperador = new PilaVec();
    this.pilaPostfija = new PilaVec();
    this.entrada = new StringTokenizer(expresion, "+*-/^() ", true);
    this.pilaOperador.apilar(new Integer(EOL));
}
/**
 * Rutina pública que realiza la evaluación. Examina la
 * maquina postfija para ver si aparece un único resultado,
 * y si es así, lo devuelve; en otro caso, se produce un error
 * @return el resultado
 */

public long obtenerValor() {
    long elResultado = 0;

    do {
        ultimoToken = obtenerToken();
        procesarToken();
    } while (ultimoToken != EOL);

    try {
        elResultado = cimaYDesapilarPostfija();
    } catch (DesbordamientoInferior e) {
        System.err.println("¡Falta un operando!");
        return 0;
    }

    if (!pilaPostfija.esVacia())
        System.err.println("Aviso: falta un operador!");

    return elResultado;
}
/**
 * Método interno que oculta la conversión de tipos.
 */
private long cimaYDesapilarPostfija() throws DesbordamientoInferior {
    return ((Long) (pilaPostfija.cimaYDesapilar())).longValue();
}

/**
 * Método interno que oculta la conversión de tipos.
 */
private int cimaPilaOper() throws DesbordamientoInferior {
    return ((Integer) (pilaOperador.cima())).intValue();
}
```

```java
    /**
     * Encuentra el siguiente token, saltando blancos, y lo devuelve.
     * Para un token VALOR, coloca el valor en valorActual.
     * Imprime un mensaje de error si no se reconoce la entrada.
     */
    private int obtenerToken() {
        String s = " ";
        try {
            s = entrada.nextToken();
        } catch (NoSuchElementException e) {
            return EOL;
        }

        if (s.equals(" "))
            return obtenerToken();
        if (s.equals("^"))
            return EXP;
        if (s.equals("/"))
            return DIV;
        if (s.equals("*"))
            return MULT;
        if (s.equals("("))
            return PAREN_A;
        if (s.equals(")"))
            return PAREN_C;
        if (s.equals("+"))
            return MAS;
        if (s.equals("-"))
            return MENOS;

        try {
            valorActual = Long.parseLong(s);
        } catch (NumberFormatException e) {
            System.err.println("Error");
            return EOL;
        }
        return VALOR;
    }
```

```java
/**
 * Después de leer un token, se utiliza el algoritmo de análisis de
 * expresiones con precedencia de operadores para evaluarlo.
 * Se detecta la falta de paréntesis abiertos.
 */
private void procesarToken() {
    int opCima;

    try {
        switch (ultimoToken) {
            case VALOR :
                pilaPostfija.apilar(new Long(valorActual));
                return;
            case PAREN_C :
                while ((opCima = cimaPilaOper()) != PAREN_A && opCima != EOL)
                    opBinario(opCima);
                if (opCima == PAREN_A)
                    pilaOperador.desapilar(); //Eliminar paréntesis abierto
                else
                    System.err.println("Falta paréntesis abierto");
                break;
            default : //Caso de operador normal
                while (tablaPrec[ultimoToken].simboloEntrada
                        <= tablaPrec[opCima = cimaPilaOper()].cimaPila)
                    opBinario(opCima);
                if (ultimoToken != EOL)
                    pilaOperador.apilar(new Integer(ultimoToken));
                break;
        }
    } catch (DesbordamientoInferior e) {
    } //No puede ocurrir
}
```

```java
/**
 * cimaYDesapilar de la pila de la máquina postfija; devuelve el
 * resultado. Si la pila está vacia, se genera un error
 */
private long obtenerCima() {
    try {
        return cimaYDesapilarPostfija();
    } catch (DesbordamientoInferior e) {
        System.err.println("Falta un operando");
    }
    return 0;
}
/**
 * Procesa un operador tomando dos elementos de la pila
 * postfija, aplicando el operador, y apilando el resultado.
 * Imprime un error si faltan paréntesis derechos o se divide por 0.
 */
private void opBinario(int opCima) {

    if (opCima == PAREN_A) {
        System.err.println("Paréntesis desequilibrados");
        try {
            pilaOperador.desapilar();
        } catch (DesbordamientoInferior e) {} //No puede ocurrir
        return;
    }
    long lder = obtenerCima();
    long lizq = obtenerCima();
    if (opCima == EXP)
        pilaPostfija.apilar(new Long(potenciaRec(lizq, lder)));
    else if (opCima == MAS)
        pilaPostfija.apilar(new Long(lizq + lder));
    else if (opCima == MENOS)
        pilaPostfija.apilar(new Long(lizq - lder));
    else if (opCima == MULT)
        pilaPostfija.apilar(new Long(lizq * lder));
    else if (opCima == DIV)
        if (lder != 0)
            pilaPostfija.apilar(new Long(lizq / lder));
        else {
            System.err.println("Division por cero");
            pilaPostfija.apilar(new Long(lizq));
        }
    try {
        pilaOperador.desapilar();
    } catch (DesbordamientoInferior e) {}
}
```

```
    private static long potenciaRec(long x, long n) {
        if (x == 0) {
            if (n == 0)
                System.err.println("0^0 es indeterminado");
            return 0;
        }
        if (n < 0) {
            System.err.println("exponente negativo");
            return 0;
        }
        if (n == 0)
            return 1;
        if (n % 2 == 0)
            return potenciaRec(x * x, n / 2);
        else
            return x * potenciaRec(x, n - 1);
    }
}
```

Análisis de la Potencia recursiva

- si el exponente es par, el problema tiene una evolución logarítmica
- si es impar, su evolución es lineal.

No obstante, como si "n" es impar entonces "n-1" es par, el caso peor es que en la mitad de los casos tengamos "n" impar y en la otra mitad sea par. El caso mejor, por contra, es que siempre sea "n" par.

Un ejemplo de caso peor seria x^{31}, que implica la siguiente serie para n:

31 30 15 14 7 6 3 2 1

cuyo número de términos podemos acotar superiormente por 2 * eis ($\log_2(n)$), donde eis(r) es el entero inmediatamente superior.

Por tanto, la complejidad de la Potencia recursiva es de orden **O(log n)**.

Prueba calculadora

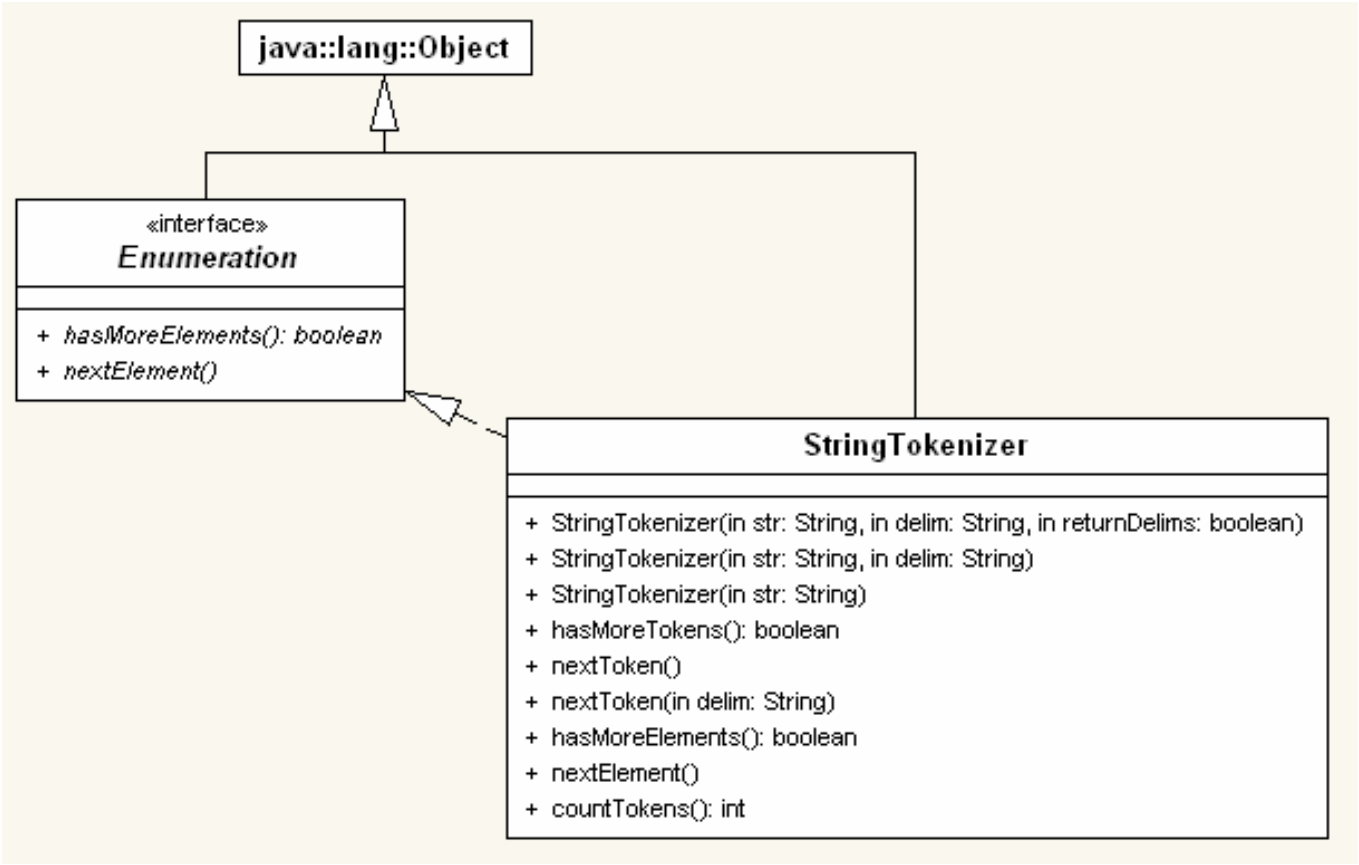
```
public class PruebaCalculadora {
    public static void main(String[] args) {
        String cadena;
        BufferedReader in =
            new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.println("Introduzca expresiones, una por línea");
            while ((cadena = in.readLine()).length() != 0) {
                System.out.println("Leída: " + cadena);
                Evaluador evaluador = new Evaluador(cadena);
                System.out.println(evaluador.obtenerValor());
                System.out.println("Introduzca otra expresión;");
            }
        } catch (IOException e) {
        }
    }
}
```

==========

```
Introduzca expresiones, una por línea
2+2
Leída: 2+2
4
Introduzca otra expresión;
((2+2))
Leída: ((2+2))
4
Introduzca otra expresión;
2*(2^2)
Leída: 2*(2^2)
8
Introduzca otra expresión;
((2-1)
Leída: ((2-1)Paréntesis desequilibrados

1
Introduzca otra expresión;
```

La clase StringTokenizer



java.util

Class StringTokenizer

java.lang.Object
  └ **java.util.StringTokenizer**

All Implemented Interfaces:

Enumeration<Object>

```
public class StringTokenizer
extends Object
implements Enumeration<Object>
```

The string tokenizer class allows an application to break a string into tokens. The tokenization method is much simpler than the one used by the `StreamTokenizer` class. The `StringTokenizer` methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments.

The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.

Constructor Summary

`StringTokenizer(String str)`	Constructs a string tokenizer for the specified string.
`StringTokenizer(String str, String delim)`	Constructs a string tokenizer for the specified string.
`StringTokenizer(String str, String delim, boolean returnDelims)`	Constructs a string tokenizer for the specified string.

Method Summary

int	`countTokens()` Calculates the number of times that this tokenizer's `nextToken` method can be called before it generates an exception.
boolean	`hasMoreElements()` Returns the same value as the `hasMoreTokens` method.
boolean	`hasMoreTokens()` Tests if there are more tokens available from this tokenizer's string.
Object	`nextElement()` Returns the same value as the `nextToken` method, except that its declared return value is `Object` rather than `String`.
String	`nextToken()` Returns the next token from this string tokenizer.
String	`nextToken(String delim)` Returns the next token in this string tokenizer's string.

```java
import java.util.StringTokenizer;

public class StringTokenizing {
    public static void main(String[] args) {
        String texto = "En un lugar de la mancha de cuyo nombre no quiero"
                + " acordarme vivían un hidalgo, el ama que pasaba de los "
                + "cuarenta, su sobrina que no llegaba a los veinte.";
        String delimitadores = ",.";
        StringTokenizer tokens = new StringTokenizer(texto, delimitadores);
        System.out.println("Número de tokens: " + tokens.countTokens());
        while (tokens.hasMoreTokens())
            System.out.println(tokens.nextToken());
    }
}
```

```
Número de tokens: 3
En un lugar de la mancha de cuyo nombre no quiero acordarme vivían un hidalgo
 el ama que pasaba de los cuarenta
 su sobrina que no llegaba a los veinte
```

```java
import java.util.StringTokenizer;

class EjemploStringTokenizer
{
   public static void main (String [] args)
   {
      String NumeroTelefono = "(34) 950-121212";
      StringTokenizer st = new StringTokenizer (NumeroTelefono, "()");
      System.out.println ("Numero de tokens = " + st.countTokens ());
      System.out.println ("Internacional =" + st.nextToken ());
      System.out.println ("Prefijo =" + st.nextToken (") -"));
      //System.out.println ("Prefijo =" + st.nextToken (")-"));
      System.out.println ("Número =" + st.nextToken ());
   }
}
```

```
Numero de tokens = 2
Internacional =34
Prefijo =950
Número =121212
```

```
Numero de tokens = 2
Internacional =34
Prefijo = 950
Número =121212
```