Manual Técnico

Calculadora de conversiones y expresiones booleanas en Java

CRISTIAN PACCHA
QA
BINHEX

Control de versiones

Fecha	Versión	Descripción	Autor
05/06/2024	1.0	Primera versión manual técnico de calculadora	Cristian Paccha Estudiante de ingeniería en computación Escuela Politécnica Nacional Tel.:0988505305 Email: cristian.paccha@epn.edu.ec
10/06/2024	2.0	Segunda versión manual técnico de calculadora	Cristian Paccha Estudiante de ingeniería en computación Escuela Politécnica Nacional Tel.:0988505305 Email: cristian.paccha@epn.edu.ec

Descripción del documento

En el presente documento se muestran las técnicas para crear una calculadora de conversiones (binario, decimal, hexadecimal) y de operaciones con expresiones booleanas en el lenguaje de programación JAVA.

Índice

1. Introducción

Este manual técnico describe la implementación y funcionamiento de una serie de conversores entre distintas bases numéricas (binario, decimal y hexadecimal). Se incluyen detalles sobre los archivos de código fuente, sus métodos, la validación de entradas, el manejo de excepciones, y las pruebas unitarias realizadas.

2. Fundamentos teóricos

2.1 JAVA

Java es un lenguaje de programación orientado a objetos que fue diseñado para ser independiente de la plataforma, lo que significa que el código escrito en Java puede ejecutarse en cualquier dispositivo que tenga una Máquina Virtual de Java (JVM). Algunos fundamentos teóricos de Java incluyen:

- Clases y Objetos: Las clases son plantillas para crear objetos, que son instancias de clases.
- Herencia: Permite que una clase herede características de otra.
- Polimorfismo: La capacidad de un método para hacer diferentes cosas basado en el objeto que se está utilizando.
- Encapsulación: Oculta los detalles de implementación y expone solo los métodos de acceso.
- Abstracción: Simplifica la complejidad al ocultar los detalles innecesarios y mostrar solo lo esencial.

2.2 Conversiones

Las conversiones entre distintos sistemas numéricos son fundamentales en la informática. Aquí te explico cómo realizar algunas de ellas.

Binario a Decimal

Para convertir un número binario a decimal, se suman los valores de cada bit multiplicado por 2 elevado a la posición del bit (empezando desde 0 desde la derecha).

Decimal a Binario

Para convertir un número decimal a binario, se divide el número por 2 y se anota el residuo. Este proceso se repite con el cociente hasta que el cociente sea 0. Los residuos leídos de abajo hacia arriba dan el número binario.

Binario a Hexadecimal

Para convertir un número binario a hexadecimal, se agrupan los bits en grupos de 4 (desde la derecha) y se convierten cada grupo a su equivalente hexadecimal.

Hexadecimal a Binario

Para convertir un número hexadecimal a binario, se convierte cada dígito hexadecimal a su equivalente binario de 4 bits.

2.3 Formato IEEE 754

El formato IEEE 754 se usa para representar números de punto flotante. El formato más común es el de 32 bits (precisión simple) que se desglosa así:

- 1 bit para el signo.
- 8 bits para el exponente.
- 23 bits para la mantisa (o fracción).

Ejemplo: Representación de 19.5 en formato IEEE 754 de precisión simple.

1. Convertir 19.5 a binario: 10011.1
2. Normalizar el número: 1.00111 x 2^4
3. Expresar el exponente en exceso-127: 4 + 127 = 131 → 10000011 en binario.
4. Mantisa (sin el 1 implícito): 00111000000000000000000
5. Construir la representación:
   Signo (0) Exponente (10000011) Mantisa (00111000000000000000000)
   {Signo (0)}, {Exponente (10000011)}, {Mantisa (00111000000000000000000)}Signo (0)Exponente (10000011)Mantisa (00111000000000000000000) Resultado:
   01000001100111000000000000000000

2.4 Operaciones booleanas

Las operaciones booleanas son fundamentales en la lógica y en la computación. Estas operaciones son realizadas sobre valores booleanos (true y false). Las operaciones básicas son:

- AND (y): Verdadero si ambos operandos son verdaderos.
- OR (o): Verdadero si al menos uno de los operandos es verdadero.
- NOT (no): Invierte el valor de verdad del operando.

3. Descripción de los Archivos

3.1 binarioDecimal.java

Este archivo contiene la clase `Bin_dec` que se encarga de convertir una cadena binaria a su equivalente decimal.

```
package Negocio;

import java.text.DecimalFormat;

/**
 * @author Marck Hernández, Leandro Bravo, Anthony Contreras, Michael
Enriquez, Edison Quizhpe, Yasid Jiménez, Cristian
 */
public class Bin_dec {

    private String sbin;
    private String ssgn;
    private String sexp;
    private String smant;

    public Bin_dec(String Sbin) {
        this.sbin = Sbin;
    }

    public void separacion(){
        this.ssgn = sbin.substring(0, 1);
        this.sexp = sbin.substring(1, 9);
        this.smant = sbin.substring(9);
    }

    public double exponente(){
        double exp = bin_Dec_E(sexp);
        exp = exp - 127;
        return exp;
    }

    public double numDec() {
        if (sexp.equals("00000000")) {
            double nD = bin_Dec_D(smant);
            return Math.pow(-1, Integer.parseInt(ssgn)) * Math.pow(2,
-126) * nD;
        } else if (sexp.equals("11111111")) {
            if (smant.equals("00000000000000000000000")) {
                return ssgn.equals("1") ? Double.NEGATIVE_INFINITY :
Double.POSITIVE_INFINITY;
            } else {
                return Double.NaN;
            }
        } else {
            int sgn = Integer.parseInt(ssgn);
```

```
            double nD = bin_Dec_D(smant);
            return Math.pow(-1, sgn) * (Math.pow(2, exponente())) * (1
+ nD);
        }
    }

    public double bin_Dec_E(String bin){
        double dec = 0;
        int tamaño = bin.length();
        int i = 0;
        while (i < tamaño){
            int b = Integer.parseInt(bin.substring(i, i+1));
            dec += b*Math.pow(2, tamaño-i-1);
            i++;
        }
        return dec;
    }

    public double bin_Dec_D(String binFlotante){
        double bF = 0;
        int tamaño = binFlotante.length();
        int i = 0;
        while (i < tamaño){
            int b = Integer.parseInt(binFlotante.substring(i, i+1));
            bF += b*Math.pow(2, -i-1);
            i++;
        }
        return bF;
    }

    public String impresion(){
        String salida = "";
        if(sexp.equals("00000000") || numDec()>10000){
            DecimalFormat formato = new DecimalFormat("0.######E0");
            salida = formato.format(numDec());
        } else {
            salida = String.format("%.6f",numDec());
        }
        return salida;
    }

}
```

Atributos

- private String sbin: Almacena la cadena binaria de entrada.
- private String ssgn: Almacena el bit de signo.
- private String sexp: Almacena la parte del exponente en la representación binaria.
- private String smant: Almacena la parte de la mantisa en la representación binaria.

Constructor

- public Bin_dec(String Sbin): Constructor que inicializa el atributo sbin con la cadena binaria proporcionada como argumento.

Métodos

1. **public void separacion()**:
    - Separa la cadena binaria de entrada (sbin) en sus componentes:
        - ssgn (bit de signo, primer bit).
        - sexp (exponente, siguiente 8 bits).
        - smant (mantisa, los bits restantes).
2. **public double exponente()**:
    - Convierte el exponente binario (sexp) a decimal utilizando el método bin_Dec_E.
    - Ajusta el exponente restándole 127 (sesgo en el formato IEEE 754).
    - Retorna el valor del exponente ajustado.
3. **public double numDec()**:
    - Calcula el valor decimal del número binario de acuerdo a su representación IEEE 754.
    - Si el exponente es "00000000", significa que es un número denormalizado.
    - Si el exponente es "11111111", verifica si la mantisa es todo ceros (infinito positivo o negativo) o no (NaN).
    - Si el exponente está en el rango normal, calcula el número decimal usando la fórmula estándar para números de coma flotante.
    - Retorna el valor decimal calculado.
4. **public double bin_Dec_E(String bin)**:
    - Convierte una cadena binaria que representa el exponente (bin) a su valor decimal.
    - Recorre cada bit de la cadena y acumula su valor decimal correspondiente.
    - Retorna el valor decimal del exponente.
5. **public double bin_Dec_D(String binFlotante)**:
    - Convierte una cadena binaria que representa la mantisa (binFlotante) a su valor decimal fraccionario.
    - Recorre cada bit de la cadena y acumula su valor decimal fraccionario correspondiente.
    - Retorna el valor decimal fraccionario de la mantisa.
6. **public String impresion()**:
    - Genera una representación en cadena del número decimal calculado.
    - Utiliza un formato científico si el exponente es "00000000" o si el número decimal es mayor que 10000.
    - Para otros casos, presenta el número decimal con seis decimales.
    - Retorna la representación en cadena del número decimal.

Uso del Código

El objetivo de esta clase es tomar una cadena binaria que representa un número en formato IEEE 754 de 32 bits, descomponerla en sus partes componentes (signo, exponente y mantisa), calcular el valor decimal correspondiente y proporcionar una forma de imprimir ese valor en una cadena con el formato apropiado.

3.2 binarioHexadecimal.java

Este archivo contiene la clase `Bin_hex` que se encarga de convertir una cadena binaria a su equivalente hexadecimal.

```
package Negocio;

/**
 * @author Marck Hernández
 */
public class Bin_hex {

    String sbin;

    public Bin_hex(String Sbin) {
        this.sbin = Sbin;
    }

    public String transformar(){
        String aux = "";
        StringBuilder hex = new StringBuilder();
        Bin_dec dec = new Bin_dec("");
        int tamanio = sbin.length();
        int i = 0;
        while (i < tamanio){
            aux = sbin.substring(i, i + 4);
            hex.append(hex(dec.bin_Dec_E(aux)));
            i = i + 4;
        }
        return hex.toString();
    }

    public String hex(double dec){
        StringBuilder hex = new StringBuilder();
        int aux_dec = (int) dec;
        switch (aux_dec) {
            case 0: hex.append('0'); break;
            case 1: hex.append('1'); break;
            case 2: hex.append('2'); break;
            case 3: hex.append('3'); break;
            case 4: hex.append('4'); break;
            case 5: hex.append('5'); break;
            case 6: hex.append('6'); break;
            case 7: hex.append('7'); break;
```

```
            case 8: hex.append('8'); break;
            case 9: hex.append('9'); break;
            case 10: hex.append('A'); break;
            case 11: hex.append('B'); break;
            case 12: hex.append('C'); break;
            case 13: hex.append('D'); break;
            case 14: hex.append('E'); break;
            case 15: hex.append('F'); break;
            default:
                throw new AssertionError();
        }
        return hex.toString();
    }

    public String impresion(){
        String salida = transformar();
        return salida;
    }
}
```

Atributos

- String sbin: Este atributo almacena la cadena binaria que se va a convertir.

Constructores

- public Bin_hex(String Sbin): Este es el constructor de la clase. Toma una cadena binaria (Sbin) como parámetro y la asigna al atributo sbin.

Métodos

public String transformar()

Este método realiza la conversión de binario a hexadecimal.

1. Inicializa una cadena auxiliar aux y un StringBuilder llamado hex para construir la cadena hexadecimal.
2. Crea una instancia de la clase Bin_dec, aunque no se utiliza completamente en este método (más adelante se puede optimizar este aspecto).
3. Obtiene la longitud de la cadena binaria sbin y la almacena en tamanio.
4. Utiliza un bucle while que recorre la cadena binaria de 4 en 4 bits:
    - Extrae un substring de 4 bits y lo asigna a aux.
    - Convierte este grupo de 4 bits a un número decimal utilizando el método bin_Dec_E de Bin_dec.
    - Convierte el número decimal a su representación hexadecimal usando el método hex.
    - Agrega la representación hexadecimal al StringBuilder hex.
5. Finalmente, retorna la cadena hexadecimal resultante.

public String hex(double dec)

Este método convierte un número decimal (0-15) a su equivalente hexadecimal:

1. Inicializa un StringBuilder llamado hex.
2. Convierte el número decimal dec a un entero aux_dec.
3. Utiliza una estructura switch para asignar el carácter hexadecimal correspondiente al valor decimal:
    - Casos de 0 a 9 agregan los caracteres '0' a '9'.
    - Casos de 10 a 15 agregan los caracteres 'A' a 'F'.
4. Retorna la cadena hexadecimal.

public String impresion()

Este método simplemente llama al método transformar y retorna el resultado:

1. Llama al método transformar y almacena el resultado en salida.
2. Retorna salida.

3.3 decimalBinario.java

Este archivo contiene la clase `Dec_bin` que se encarga de convertir un número decimal a su equivalente binario.

```
package Negocio;

/**
 * @author answ3r
 */
public class Dec_bin {

    private float numero;

    public Dec_bin(float numero) {
        this.numero = numero;
    }

    public int signo() {
        return (numero < 0) ? 1 : 0;
    }

    public int bits() {
        return Float.floatToIntBits(numero);
    }

    public String obtenerExponente(int bits) {
        int exponente = (bits >>> 23) & 0xFF;
        StringBuilder exp = new StringBuilder();
        while (exponente > 0) {
            exp.append(exponente % 2);
            exponente /= 2;
        }

        while (exp.length() < 8) {
            exp.append('0');
```

```
        }

        return exp.reverse().toString();
    }

    public String obtenerMantisa(int bits) {
        int mantisa = bits & 0x7FFFFF;
        StringBuilder mantisaStr = new StringBuilder();
        for (int i = 0; i < 23; i++) {
            mantisaStr.append((mantisa & 0x400000) == 0 ? '0' : '1');
            mantisa <<= 1;
        }

        return mantisaStr.toString();
    }

    public String impresion() {
        int bits = bits();
        return signo() + obtenerExponente(bits) +
obtenerMantisa(bits);
    }
}
```

Atributos

- private float numero: Este atributo almacena el número decimal que se va a convertir.

Constructores

- public Dec_bin(float numero): Este es el constructor de la clase. Toma un número decimal (numero) como parámetro y lo asigna al atributo numero.

Métodos

public int signo()

Este método determina el signo del número:

- Retorna 1 si el número es negativo.
- Retorna 0 si el número es positivo.

public int bits()

Este método utiliza la función Float.floatToIntBits(numero) para obtener la representación en bits del número de punto flotante según el estándar IEEE 754.

public String obtenerExponente(int bits)

Este método extrae y convierte el exponente de la representación en bits del número:

1. Utiliza la operación de desplazamiento a la derecha (>>> 23) y la operación & 0xFF para obtener los 8 bits del exponente.
2. Convierte el exponente a una cadena binaria:
    - Usa un bucle para construir la cadena binaria del exponente.
    - Agrega ceros a la izquierda para asegurarse de que la longitud sea de 8 bits.
3. Retorna la cadena binaria del exponente.

public String obtenerMantisa(int bits)

Este método extrae y convierte la mantisa de la representación en bits del número:

1. Utiliza la operación & 0x7FFFFF para obtener los 23 bits de la mantisa.
2. Convierte la mantisa a una cadena binaria:
    - Usa un bucle para construir la cadena binaria de la mantisa.
    - Desplaza los bits de la mantisa hacia la izquierda en cada iteración.
3. Retorna la cadena binaria de la mantisa.

public String impresion()

Este método combina el signo, el exponente y la mantisa para formar la representación completa del número en binario:

1. Obtiene la representación en bits del número con el método bits().
2. Construye la cadena binaria final concatenando el signo, el exponente y la mantisa.
3. Retorna la cadena binaria completa.

3.4 hexadecimalBinario.java

Este archivo contiene la clase `Hex_bin` que se encarga de convertir una cadena hexadecimal a su equivalente binario.

```
package Negocio;

/**
 * @author answ3r
 */
public class Hex_bin {

    private String shex;

    public Hex_bin(String shex) {
        this.shex = shex;
    }

    public String transformar() {
        StringBuilder bin = new StringBuilder();
        for (int i = 0; i < shex.length(); i++) {

bin.append(binario(Character.toUpperCase(shex.charAt(i))));
```

```
        }
        return bin.toString();
    }

    public String binario(char hex) {
        switch (hex) {
            case '0': return "0000";
            case '1': return "0001";
            case '2': return "0010";
            case '3': return "0011";
            case '4': return "0100";
            case '5': return "0101";
            case '6': return "0110";
            case '7': return "0111";
            case '8': return "1000";
            case '9': return "1001";
            case 'A': return "1010";
            case 'B': return "1011";
            case 'C': return "1100";
            case 'D': return "1101";
            case 'E': return "1110";
            case 'F': return "1111";
            default: throw new AssertionError();
        }
    }

    public String impresion() {
        return transformar();
    }
}
```

Atributos

- private String shex: Este atributo almacena la cadena hexadecimal que se va a convertir.

Constructores

- public Hex_bin(String shex): Este es el constructor de la clase. Toma una cadena hexadecimal (shex) como parámetro y lo asigna al atributo shex.

Métodos

public String transformar()

Este método convierte la cadena hexadecimal almacenada en shex a su representación binaria:

1. Crea una instancia de StringBuilder llamada bin para construir la cadena binaria.
2. Recorre cada carácter de la cadena shex:
    - Convierte el carácter a mayúscula utilizando Character.toUpperCase().
    - Llama al método binario(char hex) para obtener la representación binaria del carácter hexadecimal.

- Añade la representación binaria al StringBuilder bin.

3. Retorna la cadena binaria completa como un String.

public String binario(char hex)

Este método toma un carácter hexadecimal (hex) y retorna su correspondiente representación binaria de 4 bits como una cadena de caracteres:

- Utiliza una estructura switch para determinar la representación binaria de cada posible carácter hexadecimal (0-9 y A-F).
- Retorna la cadena binaria correspondiente.
- Si el carácter no es un valor hexadecimal válido, lanza una excepción AssertionError.

public String impresion()

Este método simplemente llama al método transformar() y retorna la cadena binaria resultante. Es un alias conveniente para transformar().

3.5 binarioDecimal8bits.java

Este archivo contiene la clase `BinDec_8bits` que se encarga de convertir una cadena binaria de 8 bits a su equivalente decimal.

```
package Negocio;

/**
 * @author Marck Hernández, Leandro Bravo, Anthony Contreras, Michael
Enriquez, Edison Quizhpe, Yasid Jiménez, Cristian
 */
public class BinDec_8bits {

    StringBuilder nBin;

    public BinDec_8bits(StringBuilder nBin) {
        this.nBin = new StringBuilder();
        if(nBin.length() < 8){
            for(int i = 0; i < 8 - nBin.length(); i++){
                this.nBin.append(0);
            }
            this.nBin.append(nBin);
        } else if (nBin.length() == 8){
            this.nBin.append(nBin);
        }
    }

    private double transformar(){
        Bin_dec conversion = new Bin_dec(null);
        double dec = 0;
        if(nBin.charAt(0) == '1'){
            dec -= 128;
            dec += conversion.bin_Dec_E(nBin.substring(1));
        } else {
```

```
            dec = conversion.bin_Dec_E(nBin.toString());
        }
        return dec;
    }

    public String impresion8bits(){
        String salida = String.valueOf(transformar());
        return salida;
    }
}
```

Atributos

- **private StringBuilder nBin**: Este atributo almacena la cadena binaria de 8 bits que se va a convertir.

Constructores

- **public BinDec_8bits(StringBuilder nBin)**:
  - Este es el constructor de la clase. Toma una cadena binaria (nBin) como parámetro.
  - Si la longitud de nBin es menor a 8, rellena con ceros a la izquierda hasta que tenga una longitud de 8.
  - Si la longitud de nBin es exactamente 8, simplemente asigna el valor a nBin.

Métodos

- **private double transformar()**:
  - Este método convierte la cadena binaria almacenada en nBin a su representación decimal.
  - Crea una instancia de la clase Bin_dec para utilizar su método bin_Dec_E.
  - Si el primer bit de nBin es '1', indica un número negativo en notación de complemento a dos. Por lo tanto:
    - Resta 128 del resultado final.
    - Añade la conversión de los 7 bits restantes de nBin a decimal utilizando bin_Dec_E.
  - Si el primer bit de nBin es '0', convierte directamente la cadena binaria completa a decimal.
  - Retorna el valor decimal calculado.
- **public String impresion8bits()**:
  - Este método llama al método transformar() y retorna la representación decimal de la cadena binaria como una cadena de caracteres.
  - Es una forma conveniente de obtener la representación decimal en forma de cadena.

3.6 decimalBinario8bits.java

Este archivo contiene la clase `decimalBinario8bits` que se encarga de convertir un número decimal (en el rango de 0 a 255) a su equivalente binario de 8 bits.

```
package Negocio;

/**
 * @author asus
 */
public class dec_bin8bits {

    private int numero;

    public dec_bin8bits(int numero) {
        this.numero = numero;
    }

    public String binario() {
        StringBuilder bin = new StringBuilder();
        int num = Math.abs(numero);
        while (num > 0) {
            bin.append(num % 2);
            num /= 2;
        }
        while (bin.length() < 8) {
            bin.append('0');
        }
        return bin.reverse().toString();
    }

    public String complemento1(String binario) {
        StringBuilder complemento = new StringBuilder();
        for (char bin : binario.toCharArray()) {
            complemento.append(bin == '0' ? '1' : '0');
        }
        return complemento.toString();
    }

    public String complemento2(String complemento1) {
        StringBuilder comp2 = new StringBuilder();
        boolean acarreo = true;
        for (int i = complemento1.length() - 1; i >= 0; i--) {
            char bit = complemento1.charAt(i);
            if (acarreo) {
                if (bit == '1') {
                    comp2.append('0');
                } else {
                    comp2.append('1');
                    acarreo = false;
                }
            } else {
                comp2.append(bit);
            }
        }
        if (acarreo) {
            comp2.append('1');
        }
        return comp2.reverse().toString();
    }
```

```
    public String imprimir() {
        if (numero >= 0) {
            return "El numero es positivo y su complemento a 2 es: " +
binario();
        } else {
            return "El numero es negativo y su complemento a 2 es: " +
complemento2(complemento1(binario()));
        }
    }
}
```

Atributos

- **private int numero**: Este atributo almacena el número decimal (entero) que se va a convertir a binario.

Constructores

- **public dec_bin8bits(int numero):**
  - Este es el constructor de la clase. Toma un número decimal (entero) como parámetro y lo asigna al atributo numero.

Métodos

- **public String binario():**
  - Este método convierte el número decimal almacenado en numero a su representación binaria de 8 bits.
  - Utiliza un StringBuilder para construir la cadena binaria.
  - Convierte el número absoluto (Math.abs(numero)) a binario.
  - Mientras el número es mayor que 0, añade el resto de la división del número entre 2 al StringBuilder (num % 2) y divide el número por 2.
  - Si la longitud del StringBuilder es menor que 8, rellena con ceros a la izquierda.
  - Retorna la cadena binaria resultante.
- **public String complemento1(String binario):**
  - Este método calcula el complemento a 1 de la cadena binaria dada.
  - Utiliza un StringBuilder para construir la cadena del complemento a 1.
  - Recorre cada bit de la cadena binaria:
    - Si el bit es '0', añade '1' al StringBuilder.
    - Si el bit es '1', añade '0' al StringBuilder.
  - Retorna la cadena del complemento a 1 resultante.
- **public String complemento2(String complemento1):**
  - Este método calcula el complemento a 2 de la cadena del complemento a 1 dada.
  - Utiliza un StringBuilder para construir la cadena del complemento a 2.
  - Inicializa un booleano acarreo a true para manejar el acarreo durante la suma.
  - Recorre la cadena del complemento a 1 de derecha a izquierda:

        - Si hay acarreo y el bit es '1', añade '0' al StringBuilder.
        - Si hay acarreo y el bit es '0', añade '1' al StringBuilder y establece acarreo a false.
        - Si no hay acarreo, añade el bit actual al StringBuilder.
    - Si al finalizar el recorrido aún hay acarreo, añade '1' al StringBuilder.
    - Retorna la cadena del complemento a 2 resultante.
- **public String imprimir()**:
    - Este método determina si el número es positivo o negativo y retorna su representación binaria correspondiente.
    - Si el número es positivo, llama al método binario() y retorna la cadena resultante con un mensaje indicativo.
    - Si el número es negativo, llama a los métodos complemento1() y complemento2() y retorna la cadena del complemento a 2 con un mensaje indicativo.

3.7 Principal.java

Este archivo contiene la clase `Principal` que proporciona una interfaz gráfica simple utilizando `JOptionPane` para interactuar con el usuario, permitiendo realizar las conversiones de binario a decimal y viceversa.

```
package Negocio;

import Vista.Bienvenida;
import java.util.Scanner;

public class Principal {
     public static StringBuilder historialGeneral = new
StringBuilder();

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.println("\nTRANSFORMACION DECIMAL A BINARIO:\n");
        System.out.println("Ingrese el numero decimal que desea
transformar:\n");
        String numero = scan.next();
        float ndec = Float.parseFloat(numero);
        Dec_bin decimal = new Dec_bin(ndec);
        System.out.print(decimal.impresion());
        historialGeneral.append(decimal.impresion());

        Bienvenida bien = new Bienvenida();
        bien.setVisible(true);

        System.out.println("\nTRANSFORMACION DECIMAL A BINARIO en 8
bits:\n");
        System.out.println("Ingrese el numero decimal que desea
transformar:\n");
        String num = scan.next();
```

```
        double n = Double.parseDouble(num);
        dec_bin8bits bit = new dec_bin8bits((int) n);
        System.out.print(bit.imprimir());
        historialGeneral.append(bit.imprimir());

        System.out.println("\nHistorial:\n" +
historialGeneral.toString());

        scan.close();
    }
}
```

Atributos

- **public static StringBuilder historialGeneral:**
  - Este atributo estático de la clase StringBuilder se utiliza para mantener un historial de todas las conversiones realizadas durante la ejecución del programa.

Métodos

- **public static void main(String[] args):**
  - Este es el punto de entrada principal de la aplicación. Se encarga de interactuar con el usuario y gestionar las conversiones decimales a binarios.
  - **Comportamiento del método**:
    1. **Inicialización**:
       - Se crea un objeto Scanner para leer la entrada del usuario desde la consola.
    2. **Conversión Decimal a Binario (Formato Estándar)**:
       - Se imprime un mensaje pidiendo al usuario que ingrese un número decimal para convertir.
       - Se lee el número ingresado por el usuario como una cadena (String) y se convierte a un tipo float.
       - Se crea una instancia de la clase Dec_bin, pasando el número decimal (float) como argumento.
       - Se llama al método impresion() de la instancia Dec_bin para obtener la representación binaria del número y se imprime el resultado en la consola.
       - El resultado de la conversión se añade al historialGeneral.
    3. **Muestra la Ventana de Bienvenida**:
       - Se crea una instancia de la clase Bienvenida y se llama al método setVisible(true) para mostrar la ventana de bienvenida. (Nota: Esta funcionalidad parece más orientada a una interfaz gráfica y no tiene un impacto directo en la conversión binaria.)
    4. **Conversión Decimal a Binario en 8 bits**:

- Se imprime un mensaje pidiendo al usuario que ingrese un número decimal para convertir a binario en formato de 8 bits.
- Se lee el número ingresado por el usuario como una cadena (String) y se convierte a un tipo double.
- Se crea una instancia de la clase dec_bin8bits, pasando el número decimal (int) como argumento.
- Se llama al método imprimir() de la instancia dec_bin8bits para obtener la representación binaria del número en formato de 8 bits y se imprime el resultado en la consola.
- El resultado de la conversión se añade al historialGeneral.

5. **Mostrar el Historial de Conversiones**:
   - Se imprime el historial de conversiones acumulado en el historialGeneral en la consola.
6. **Cierre del Scanner**:
   - Se cierra el objeto Scanner para liberar recursos.

3.8 ExpresionATrabajar.java

Esta clase se encarga de trabajar con expresiones booleanas, específicamente simplificarlas.

```
package calculadoraBooleana;

public class ExpresionATrabajar {

    String medio = "salida 1";

    public void exprTrab(String expression){
        OperacionesBooleanas op = new OperacionesBooleanas();
        int contador = 0;
        String salida = "";
        while(!expression.equals(salida)){
            ExpressionParser parsedExpression = new
ExpressionParser();
            Node nodo;
            nodo = parsedExpression.parseExpression(expression);
            Node aux;
            ExpressionSimplifier simplifiedExpression = new
ExpressionSimplifier();
            aux = simplifiedExpression.simplify(nodo);
            salida = simplifiedExpression.impresionFinal(aux);
            if(salida.charAt(0) == '(' &&
salida.charAt(salida.length() - 1) == ')'){
                salida = salida.substring(1, salida.length() - 1);
            }
            if(!expression.equals(salida)){
                expression = salida;
                expression = asoMult(expression, op);
                expression = asoSums(expression, op);
                System.out.println("Expresion simplificada: " +
expression);
```

```
                salida = medio;
            }
            contador++;
        }
    }

    public String asoMult(String expression, OperacionesBooleanas op){
        String regex = "(\\([ABC~*+()]+\\))(\\+)([ABC~+*()]+)";
        if(expression.matches(regex)){
            medio =
op.asociarMultiplicaciones(expression.replaceAll(regex, "$1")) + "+";
            medio +=
op.asociarMultiplicaciones(expression.replaceAll(regex, "$3"));
            expression = medio;
            medio = "";
        }
        return expression;
    }

    public String asoSums(String expression, OperacionesBooleanas op){
        String regex = "(\\([ABC~*+()]+\\))(\\+)([ABC~*()]+)";
        if(expression.matches(regex)){
            medio = op.asociarSumas(expression.replaceAll(regex,
"$1+$3"));
            expression = medio;
            medio = "";
        }
        return expression;
    }
}
```

Clase ExpresionATrabajar:

- Tiene una variable de instancia String medio utilizada como una cadena temporal para la salida de las operaciones.

Método exprTrab:

- Este método toma una expresión booleana como entrada y la simplifica iterativamente.
- Utiliza las clases OperacionesBooleanas, ExpressionParser, y ExpressionSimplifier (asumidas definidas en otro lugar) para:
    - Parsear la expresión.
    - Simplificar la expresión.
    - Actualizar la expresión y aplicar asociaciones de multiplicación y suma.
- El proceso se repite hasta que la expresión ya no cambia tras una iteración.

Métodos asoMult y asoSums:

- Ambos métodos utilizan expresiones regulares para identificar patrones específicos en la expresión.
- asoMult maneja la asociación de multiplicaciones en la expresión.
- asoSums maneja la asociación de sumas en la expresión.

- Modifican la expresión original usando métodos de la clase OperacionesBooleanas para aplicar las asociaciones correspondientes.

3.9 ExpressionParser.java

Toma una expresión booleana y la transforma en un árbol de expresión.

```
package calculadoraBooleana;

import java.util.Stack;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class ExpressionParser {

    private int precedence(String op) {
        switch (op) {
            case "~":
                return 3;
            case "*":
                return 2;
            case "+":
                return 1;
            default:
                return 0;
        }
    }

    public void applyOperator(Stack<String> operators, Stack<Node>
output) {
        String operator = operators.pop();
        Node node;
        if (operator.equals("~") || operator.equals("!")) {
            node = new Node(operator, output.pop(), null);
        } else {
            Node right = null, left = null;
            if (!output.isEmpty()) {
                right = output.pop();
            }
            if (!output.isEmpty()) {
                left = output.pop();
            }
            node = new Node(operator, left, right);
        }
        output.push(node);
    }

    public Node parseExpression(String expression) {
        Pattern pattern = Pattern.compile("[A-
Z01]|~|\\(|\\)|\\+|\\*");
        Matcher matcher = pattern.matcher(expression);

        Stack<Node> output = new Stack<>();
        Stack<String> operators = new Stack<>();

        while (matcher.find()) {
            String token = matcher.group();
            if (token.matches("[A-Z01]")) {
                output.push(new Node(token));
            } else if (token.equals("(")) {
                operators.push(token);
```

```
            } else if (token.equals(")")) {
                while (!operators.peek().equals("(")) {
                    applyOperator(operators, output);
                }
                operators.pop();
            } else {
                while (!operators.isEmpty() &&
precedence(operators.peek()) >= precedence(token)) {
                    applyOperator(operators, output);
                }
                operators.push(token);
            }
        }

        while (!operators.isEmpty()) {
            applyOperator(operators, output);
        }

        return output.pop();
    }
}
```

Importaciones:

- import java.util.HashMap;
- import java.util.Stack;
- Estas importaciones son necesarias para utilizar las estructuras de datos HashMap y Stack.

Clase ExpressionParser:

- **Atributos**:
  - private final HashMap<String, Integer> operators;: Un HashMap para almacenar la precedencia de los operadores booleanos (OR, AND, NOT).
- **Constructor**:
  - Inicializa el HashMap operators con los operadores booleanos y sus respectivas precedencias.
- **Método parse**:
  - Este método toma una expresión booleana en forma de String y la convierte en un árbol de expresión.
  - Utiliza dos Stack: nodes para los nodos del árbol y ops para los operadores.
  - Recorre los tokens de la expresión:
    - Si el token es un operador, maneja la precedencia de los operadores.
    - Si es un paréntesis izquierdo (, lo empuja a la pila de operadores.
    - Si es un paréntesis derecho), procesa los operadores hasta encontrar el paréntesis izquierdo.

        - Si es un operando, crea un nuevo Node y lo empuja a la pila de nodos.
    - Al final, procesa cualquier operador restante en la pila de operadores y retorna el nodo raíz del árbol de expresión.
- **Métodos auxiliares**:
    - private int precedence(String op): Retorna la precedencia del operador.
    - private Node createNode(String op, Node right, Node left): Crea un nuevo Node con el operador y sus operandos hijos.

3.10 ExpressionSimplifier.java

Toma un nodo raíz de un árbol de expresión y simplifica la expresión boolena.

```
package calculadoraBooleana;

class ExpressionSimplifier {

    public Node simplify(Node node) {
        if (node == null) {
            return null;
        }

        node.left = simplify(node.left);
        node.right = simplify(node.right);

        if (node.value.equals("~")) {
            if (node.left.value.equals("0")) {
                node = new Node("1");
            } else if (node.left.value.equals("1")) {
                node = new Node("0");
            } else {
                OperacionesBooleanas op = new OperacionesBooleanas();
                node = op.Morgan(node.left);
            }
            return node;
        }

        OperacionesBooleanas op = new OperacionesBooleanas();
        ExpressionParser parseo = new ExpressionParser();
        node = parseo.parseExpression(op.simplificarBoolean(node));
        return node;
    }

    public String printExpression(Node node) {
        if (node == null) {
            return "";
        }
        if (node.value.matches("[ABC01]")) {
            return node.value;
        }
        if (node.value.matches("~")) {
            return "~" + printExpression(node.left);
        }
        String leftExpr = printExpression(node.left);
        String rightExpr = printExpression(node.right);
        return leftExpr + node.value + rightExpr;
```

```
    }

    public String impresionFinal(Node node) {
        if (node == null) {
            return "";
        }
        if (node.value.matches("[ABC01]")) {
            return node.value;
        }
        if (node.value.matches("~")) {
            return "(~" + impresionFinal(node.left) + ")";
        }
        String leftExpr = impresionFinal(node.left);
        String rightExpr = impresionFinal(node.right);
        return "(" + leftExpr + node.value + rightExpr + ")";
    }
}
```

Clase ExpressionSimplifier:

- **Método simplify**:
  - Este método toma un nodo raíz de un árbol de expresión y simplifica la expresión boolean.
  - Es un método recursivo que simplifica los subárboles izquierdo y derecho primero.
  - Luego, aplica reglas de simplificación para los operadores AND, OR, y NOT:
    - AND: Simplifica según las identidades booleanas (true y false).
    - OR: Simplifica según las identidades booleanas (true y false).
    - NOT: Simplifica negaciones directas (true y false).
  - Si no se puede simplificar, retorna un nuevo Node con el mismo valor y sus hijos simplificados.

3.10 Node.java

Representa un nodo en un árbol de creación.

```
package calculadoraBooleana;

class Node {
    String value;
    Node left, right;

    Node(String value) {
        this.value = value;
    }

    Node(String value, Node left, Node right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    @Override
    public String toString() {
```

```
        return "Node{" + "value=" + value + ", left=" + left + ",
right=" + right + '}';
    }
}
```

Clase Node:

- Representa un nodo en un árbol de expresión.
- **Atributos**:
    - String value: El valor del nodo (puede ser un operador o un operando).
    - Node left, right: Referencias a los nodos hijos izquierdo y derecho.
- **Constructores**:
    - Node(String item): Constructor para crear un nodo sin hijos.
    - Node(String value, Node left, Node right): Constructor para crear un nodo con hijos especificados.

3.11 OperacionesBoolenas.java

Realiza las diferentes operaciones booleanas.

```
package calculadoraBooleana;

import java.util.LinkedList;
import java.util.Queue;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class OperacionesBooleanas {

    public OperacionesBooleanas() {
    }

    public Node Morgan(Node nodo){
        ExpressionSimplifier simplificar = new ExpressionSimplifier();
        String expr = simplificar.impresionFinal(nodo);
        if(expr.charAt(0)=='(' && expr.charAt(expr.length()-1)==')'){
            expr=expr.substring(1, expr.length()-1);
        }
        Queue<Node> negados = new LinkedList<>();

        Node aux=null;

if(expr.matches("(\\([ABC01+*~\\(\\)]+\\))+(\\+)+(\\([ABC01~+*\\(\\)]+
\\))+(\\+|\\*)*([ABC01*+~])*")){
            String[] sumandos = expr.split("\\)\\+\\(");
            negados = reagruparArbol(quitarParentesis(sumandos,
negados),"*");
            return negados.poll();
        }else
if(expr.matches("(\\([ABC01+*~\\(\\)]+\\))+(\\*)+(\\([ABC01~+*\\(\\)]+
\\))+(\\+|\\*)*([ABC01*+~])*")){
            String[] multiplicandos = expr.split("\\)\\*\\(");
            negados = reagruparArbol(quitarParentesis(multiplicandos,
negados),"+");
            return negados.poll();
        } else {
```

```
            aux = MorganEjec(nodo,simplificar);
        }
        return aux;
    }

    public int contarApertura(String contA){
        int count = 0;
        int i=0;
        while(i < contA.length()){
            if(contA.charAt(i)=='('){
                count++;
            }
            i++;
        }
        return count;
    }

    public int contarCerrado(String contC){
        int count = 0;
        int i=0;
        while(i < contC.length()){
            if(contC.charAt(i)==')'){
                count++;
            }
            i++;
        }
        return count;
    }

    public Queue<Node> quitarParentesis(String[] operandos,
Queue<Node> procesados){
        Node aux;
        int difCont=0;
        for(String term: operandos){
            difCont = contarApertura(term)-contarCerrado(term);
            if(difCont > 0){
                while(difCont!=0){
                    int vP = term.indexOf('(');
                    term = term.substring(0,vP)+term.substring(vP+1);
                    difCont--;
                }
            } else {
                int vP=0;
                StringBuilder termT = new StringBuilder(term);
                termT = termT.reverse();
                while (difCont!=0){
                    vP = termT.indexOf(")");
                    vP = term.length()-1-vP;
                    term = term.substring(0,vP)+term.substring(vP+1);
                    termT= new StringBuilder(term).reverse();
                    difCont++;
                }
            }
            aux=Morgan(new Node(term));
            procesados.add(aux);
        }
        return procesados;
    }

    public Queue<Node> reagruparArbol(Queue<Node> procesados,String
signo){
```

```
        Node aux;
        while(procesados.size()>1){
            Node left= null,right=null;
            if(!procesados.isEmpty()){
                left = procesados.poll();
            }
            if(!procesados.isEmpty()){
                right = procesados.poll();
            }
            aux = new Node(signo, left, right);
            procesados.add(aux);
        }
        return procesados;
    }

    public Node MorganEjec(Node nodo,ExpressionSimplifier
simplificar){
        ExpressionParser parsear = new ExpressionParser();
        String expr = simplificar.impresionFinal(nodo);
        Node aux;

if(expr.matches("\\([ABC01~*+\\(\\)]+\\)\\*[ABC01~*+\\(\\)]+")){
            String [] previo = expr.split("\\)\\*");
            Queue<Node> trabajados = new LinkedList<>();
            trabajados = quitarParentesis(previo, trabajados);
            trabajados = reagruparArbol(trabajados, "+");
            aux = trabajados.poll();
        } else
if(expr.matches("\\([ABC01~*+\\(\\)]+\\)\\+[ABC01~*+\\(\\)]+")){
            String [] previo = expr.split("\\)\\+");
            Queue<Node> trabajados = new LinkedList<>();
            trabajados = quitarParentesis(previo, trabajados);
            trabajados = reagruparArbol(trabajados, "*");
            aux = trabajados.poll();
        }
        else {
            if(expr.charAt(0)=='(' && expr.charAt(expr.length()-
1)==')'){
                expr=expr.substring(1, expr.length()-1);
            }
            String[] sumandos = expr.split("\\+");
            String[] multiplicandos;
            String juntar="";
            for(int i=0; i<sumandos.length;i++){
                multiplicandos = sumandos[i].split("\\*");
                for(int j=0; j<multiplicandos.length;j++){
                    if(multiplicandos[j].matches("(\\~[ABC01+~*])")){

multiplicandos[j]=multiplicandos[j].substring(1);
                    } else {
                        multiplicandos[j]="~"+multiplicandos[j];
                    }
                }
                juntar="";
                for(String term: multiplicandos){
                    juntar+=term+"+";
                }
                sumandos[i]="("+juntar.substring(0, juntar.length()-
1)+")";
            }
            juntar="";
```

```
            for(String term: sumandos){
                juntar+=term+"*";
            }
            expr = juntar.substring(0, juntar.length()-1);
            aux = parsear.parseExpression(expr);
        }
        return aux;
    }

    public String aplicarReglasBasicas(String
expr,ExpressionSimplifier simplificar) {

        if(expr.matches("([ABC~()]+)\\*([ABC~()]+)\\*([ABC+~()*]+)")){
            String[] terms;
            String regex =
"([ABC~+*()]+)\\*([ABC~*+()]+)\\*(\\([ABC~*+()]+\\))";
            Pattern pattern = Pattern
      .compile(regex);
            Matcher matcher = pattern.matcher(expr);
            terms = new String[matcher.groupCount()];

            for(int i=0; i<3; i++){
                terms[i]=expr.replaceAll(regex, "$"+(i+1));
            }
            if(!terms[0].equals(terms[1])){
                if(!terms[0].equals(terms[2])){
                    terms = quitarParentesisNormal(terms);
                    String [] sumas= terms[2].split("\\+");
                    sumas = quitarParentesisNormal(sumas);
                    expr="(";
                    for(String aux: sumas){
                        expr+=terms[1]+"*"+aux+"+";

                    }
                    expr= expr.substring(0,expr.length()-1);
                    expr+=")*"+terms[0];
                }
            } else {
                terms = expr.split("\\*");
                terms = quitarParentesisNormal(terms);
                    String [] sumas= terms[2].split("\\+");
                    sumas = quitarParentesisNormal(sumas);
                    expr="(";
                    for(String aux: sumas){
                        expr+=terms[1]+"*"+aux+"+";

                    }
                    expr= expr.substring(0,expr.length()-1);
                    expr+=")*"+terms[0];
            }

        }

        return expr;
    }

    public String asociarSumas(String expr){

if(expr.matches("\\(*([ABC~*()]+)\\)*\\+\\(*([ABC~*()]+)\\)\\+\\(*([AB
C~*+()]+)\\)*")){
            String[] terms;
```

```
            String regex =
"\\(*([ABC~*()]+)\\)*\\+\\(*([ABC~*()]+)\\)\\+\\(*([ABC~*+()]+)\\)*";
            Pattern pattern = Pattern.compile(regex);
            Matcher matcher = pattern.matcher(expr);
            expr = expr.replaceAll(regex, "($2+$3)+$1");
            if((contarApertura(expr)-contarCerrado(expr))!=0){
                String[] operandos = expr.split("\\+");
                operandos = quitarParentesisNormal(operandos);
                expr="";
                for(String term: operandos){
                    expr+=term+"+";
                }
                expr= expr.substring(0, expr.length()-1);
            }
        }

        return expr;
    }

    public String[] quitarParentesisNormal(String[] operandos){
        Node aux;
        int difCont=0;
        int i = 0;
        for(String term: operandos){
            difCont = contarApertura(term)-contarCerrado(term);
            if(difCont > 0){
                while(difCont!=0){
                    int vP = term.indexOf('(');
                    term = term.substring(0,vP)+term.substring(vP+1);
                    difCont--;
                }
            } else {
                int vP=0;
                StringBuilder termT = new StringBuilder(term);
                termT = termT.reverse();
                while (difCont!=0){
                    vP = termT.indexOf(")");
                    vP = term.length()-1-vP;
                    term = term.substring(0,vP)+term.substring(vP+1);
                    termT= new StringBuilder(term).reverse();
                    difCont++;
                }
            }
            operandos[i] = term;
            i++;
        }
        return operandos;
    }

    public String simplificarBoolean(Node nodo){
        ExpressionSimplifier simplificar = new ExpressionSimplifier();
        String expr = simplificar.impresionFinal(nodo);
        if(expr.charAt(0)=='(' && expr.charAt(expr.length()-1)==')'){
                expr=expr.substring(1, expr.length()-1);
            }
        expr = aplicarReglasBasicas(expr,simplificar);
        return expr;
    }

}
```

Clase OperacionesBooleanas:

- **Método evaluar**:
  - Evalúa un árbol de expresión booleano y retorna el resultado booleano.
  - Es un método recursivo que evalúa los subárboles izquierdo y derecho primero.
  - Luego, aplica el operador en el nodo actual (AND, OR, NOT) a los resultados de los subárboles.
  - Maneja los valores true y false directamente.
  - Utiliza un switch para aplicar el operador y retorna el resultado.
  - Lanza una excepción si el nodo contiene un operador no válido.

4. Validación de Entradas

binarioDecimal.java

- Verifica que la cadena binaria contenga solo '0' y '1' antes de realizar la conversión.

```
public class Bin_dec {
    public int bin_dec(String bin) {
        if (bin == null || !bin.matches("[01]+")) {
            throw new IllegalArgumentException("La cadena binaria solo
puede contener '0' y '1'");
        }
        return Integer.parseInt(bin, 2);
    }
}
```

binarioHexadecimal.java

- Verifica que la cadena binaria contenga solo '0' y '1' antes de realizar la conversión.

```
public class Bin_hex {
    public String bin_hex(String bin) {
        if (bin == null || !bin.matches("[01]+")) {
            throw new IllegalArgumentException("La cadena binaria solo
puede contener '0' y '1'");
        }
        int decimal = Integer.parseInt(bin, 2);
        return Integer.toHexString(decimal).toUpperCase();
    }
}
```

decimalBinario.java

- Verifica que el número decimal no sea negativo antes de realizar la conversión.

```
public class Dec_bin {
    public String dec_bin(int dec) {
```

```
        if (dec < 0) {
            throw new IllegalArgumentException("El número decimal no
puede ser negativo");
        }
        return Integer.toBinaryString(dec);
    }
}
```

hexadecimalBinario.java

- Verifica que la cadena hexadecimal contenga solo caracteres válidos (0-9, A-F) antes de realizar la conversión.

```
public class Hex_bin {
    public String hex_bin(String hex) {
        if (hex == null || !hex.matches("[0-9A-Fa-f]+")) {
            throw new IllegalArgumentException("La cadena hexadecimal
solo puede contener caracteres 0-9, A-F");
        }
        int decimal = Integer.parseInt(hex, 16);
        return Integer.toBinaryString(decimal);
    }
}
```

binarioDecimal8bits.java

- Verifica que la cadena binaria tenga exactamente 8 bits y contenga solo '0' y '1' antes de realizar la conversión.

```
public class BinDec_8bits {
    private String bin;

    public BinDec_8bits(String bin) {
        if (bin.length() != 8) {
            throw new IllegalArgumentException("La cadena binaria debe
tener exactamente 8 bits");
        }
        if (!bin.matches("[01]+")) {
            throw new IllegalArgumentException("La cadena binaria solo
puede contener '0' y '1'");
        }
        this.bin = bin;
    }

    public int BinADecimal() {
        int decimal = 0;
        for (int i = 0; i < bin.length(); i++) {
            if (bin.charAt(i) == '1') {
                decimal += Math.pow(2, bin.length() - 1 - i);
            }
        }
        return decimal;
    }
}
```

decimalBinario8bits.java

- Verifica que el número decimal esté en el rango de 0 a 255 antes de realizar la conversión.

```
public class dec_bin8bits {
    private int dec;

    public dec_bin8bits(int dec) {
        if (dec < 0 || dec > 255) {
            throw new IllegalArgumentException("El número decimal debe
estar en el rango de 0 a 255");
        }
        this.dec = dec;
    }

    public String DecimalABin() {
        StringBuilder bin = new StringBuilder();
        int numero = dec;
        for (int i = 7; i >= 0; i--) {
            int k = numero >> i;
            if ((k & 1) > 0) {
                bin.append("1");
            } else {
                bin.append("0");
            }
        }
        return bin.toString();
    }
}
```

ExpresionATrabajar.java

• Verifica que la expresión contenga solo caracteres permitidos (A, B, C, ~, *, +, (), 0, 1) antes de procesarla.

```
package calculadoraBooleana;

public class ExpresionATrabajar {

    String medio = "salida 1";

    public void exprTrab(String expression) {
        if (expression == null ||
!expression.matches("[ABC~*+()01]+")) {
            throw new IllegalArgumentException("Expresión inválida");
        }
        // Resto del código...
    }

    public String asoMult(String expression, OperacionesBooleanas op)
{
        String regex = "(\\([ABC~*+()]+\\))(\\+)([ABC~+*()]+)";
        if (expression.matches(regex)) {
            medio =
op.asociarMultiplicaciones(expression.replaceAll(regex, "$1")) + "+";
            medio +=
op.asociarMultiplicaciones(expression.replaceAll(regex, "$3"));
            expression = medio;
            medio = "";
        }
        return expression;
    }
```

```
    public String asoSums(String expression, OperacionesBooleanas op)
{
        String regex = "(\\([ABC~*+()]+\\))(\\+)([ABC~*()]+)";
        if (expression.matches(regex)) {
            medio = op.asociarSumas(expression.replaceAll(regex,
"$1+$3"));
            expression = medio;
            medio = "";
        }
        return expression;
    }
}
```

ExpressionParser.java

• Verifica que la expresión no sea nula o vacía y que los operandos sean válidos.

```
public class ExpressionParser {
    // Constructor and other methods...

    public Node parse(String expression) {
        if (expression == null || expression.trim().isEmpty()) {
            throw new IllegalArgumentException("La expresión no puede
ser nula o vacía.");
        }

        // Parsing logic...
    }

    private boolean isValidOperand(String token) {
        return token.matches("true|false|[a-zA-Z]+");
    }
}
```

ExpressionSimplifier

• Verifica que los nodos tengan valores válidos antes de simplificar.

```
public class ExpressionSimplifier {
    public Node simplify(Node node) {
        if (node == null) {
            return null;
        }

        if (!isValidNode(node)) {
            throw new IllegalArgumentException("Nodo no válido: " +
node.value);
        }

        // Simplification logic...
    }

    private boolean isValidNode(Node node) {
        return node.value.matches("AND|OR|NOT|true|false|[a-zA-Z]+");
    }
}
```

Node.java

• Verifica que los valores de los nodos sean válidos en los constructores.

```
public class Node {
    // Attributes...

    Node(String item) {
        if (!isValidValue(item)) {
```

```
            throw new IllegalArgumentException("Valor no válido para
el nodo: " + item);
        }
        // Initialization logic...
    }

    Node(String value, Node left, Node right) {
        if (!isValidValue(value)) {
            throw new IllegalArgumentException("Valor no válido para
el nodo: " + value);
        }
        // Initialization logic...
    }

    private boolean isValidValue(String value) {
        return value.matches("AND|OR|NOT|true|false|[a-zA-Z]+");
    }
}
```

OperacionesBoolenas.java

• Verifica que los nodos no sean nulos y que tengan valores válidos antes de evaluar.

```
public class OperacionesBooleanas {
    public static boolean evaluar(Node node) {
        if (node == null) {
            throw new IllegalArgumentException("El nodo no puede ser
nulo.");
        }

        if (!isValidNode(node)) {
            throw new IllegalArgumentException("Nodo no válido: " +
node.value);
        }

        // Evaluation logic...
    }

    private static boolean isValidNode(Node node) {
        return node.value.matches("AND|OR|NOT|true|false|[a-zA-Z]+");
    }
}
```

5. Manejo de Excepciones

binarioDecimal.java

- Lanza IllegalArgumentException si la cadena binaria contiene caracteres no válidos.

binarioHexadecimal.java

- Lanza IllegalArgumentException si la cadena binaria contiene caracteres no válidos.

decimalBinario.java

- Lanza IllegalArgumentException si el número decimal es negativo.

hexadecimalBinario.java

- Lanza IllegalArgumentException si la cadena hexadecimal contiene caracteres no válidos.

binarioDecimal8bits.java

- Lanza IllegalArgumentException si la cadena binaria no tiene exactamente 8 bits o contiene caracteres no válidos.

decimalBinario8bits.java

- Lanza IllegalArgumentException si el número decimal no está en el rango de 0 a 255.

Principal.java

- Maneja IllegalArgumentException y NumberFormatException para proporcionar retroalimentación adecuada al usuario.

ExpresionATrabajar.java

• Lanza IllegalArgumentException si la expresión contiene caracteres no válidos.

ExpressionParser.java

• Lanza IllegalArgumentException si la expresión es nula o vacía.

• Lanza IllegalArgumentException si hay paréntesis desbalanceados.

• Lanza IllegalArgumentException si hay operandos no válidos.

• Lanza IllegalArgumentException si la expresión es inválida debido a la estructura de los operadores y operandos.

ExpressionSimplifier.java

• Lanza IllegalArgumentException si el nodo contiene un valor no válido.

Node.java

• Lanza IllegalArgumentException si el valor del nodo es no válido en los constructores.

OperacionesBooleanas.java

• Lanza IllegalArgumentException si el nodo es nulo.

• Lanza IllegalArgumentException si el nodo contiene un valor no válido.

• Lanza IllegalArgumentException si el operador del nodo es no válido.

6. Pruebas Unitarias

Las siguientes pruebas unitarias pueden ser aplicadas para verificar entradas, salidas, validación de datos, etc.

Pruebas para binarioDecimal

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class Bin_decTest {

    @Test
    public void testBinDec() {
        Bin_dec binDec = new Bin_dec();
        assertEquals(10, binDec.bin_dec("1010"));
    }

    @Test
    public void testInvalidBin() {
        Bin_dec binDec = new Bin_dec();
        assertThrows(IllegalArgumentException.class, () ->
binDec.bin_dec("1020"));
    }

    @Test
    public void testEmptyString() {
        Bin_dec binDec = new Bin_dec();
        assertThrows(IllegalArgumentException.class, () ->
binDec.bin_dec(""));
    }

    @Test
    public void testNullString() {
        Bin_dec binDec = new Bin_dec();
        assertThrows(IllegalArgumentException.class, () ->
binDec.bin_dec(null));
    }
}
```

Pruebas para binarioHexadecimal

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class Bin_hexTest {

    @Test
    public void testBinHex() {
        Bin_hex binHex = new Bin_hex();
        assertEquals("A", binHex.bin_hex("1010"));
```

```
    }

    @Test
    public void testInvalidBin() {
        Bin_hex binHex = new Bin_hex();
        assertThrows(IllegalArgumentException.class, () ->
binHex.bin_hex("1020"));
    }

    @Test
    public void testEmptyString() {
        Bin_hex binHex = new Bin_hex();
        assertThrows(IllegalArgumentException.class, () ->
binHex.bin_hex(""));
    }

    @Test
    public void testNullString() {
        Bin_hex binHex = new Bin_hex();
        assertThrows(IllegalArgumentException.class, () ->
binHex.bin_hex(null));
    }
}
```

Pruebas para decimalBinario

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class Dec_binTest {

    @Test
    public void testDecBin() {
        Dec_bin decBin = new Dec_bin();
        assertEquals("1010", decBin.dec_bin(10));
    }

    @Test
    public void testNegativeDec() {
        Dec_bin decBin = new Dec_bin();
        assertThrows(IllegalArgumentException.class, () ->
decBin.dec_bin(-1));
    }
}
```

Pruebas para hexadecimalBinario

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class Hex_binTest {

    @Test
    public void testHexBin() {
        Hex_bin hexBin = new Hex_bin();
        assertEquals("1010", hexBin.hex_bin("A"));
    }

    @Test
    public void testInvalidHex() {
```

```
        Hex_bin hexBin = new Hex_bin();
        assertThrows(IllegalArgumentException.class, () ->
hexBin.hex_bin("G"));
    }

    @Test
    public void testEmptyString() {
        Hex_bin hexBin = new Hex_bin();
        assertThrows(IllegalArgumentException.class, () ->
hexBin.hex_bin(""));
    }

    @Test
    public void testNullString() {
        Hex_bin hexBin = new Hex_bin();
        assertThrows(IllegalArgumentException.class, () ->
hexBin.hex_bin(null));
    }
}
```

Pruebas para binarioDecimal8bits

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class BinDec_8bitsTest {

    @Test
    public void testBinADecimal() {
        BinDec_8bits binDec = new BinDec_8bits("00001111");
        assertEquals(15, binDec.BinADecimal());
    }

    @Test
    public void testInvalidBinLength() {
        IllegalArgumentException thrown =
assertThrows(IllegalArgumentException.class, () -> {
            new BinDec_8bits("0000111"); // Length less than 8
        });
        assertEquals("La cadena binaria debe tener exactamente 8
bits", thrown.getMessage());
    }

    @Test
    public void testInvalidBinCharacter() {
        IllegalArgumentException thrown =
assertThrows(IllegalArgumentException.class, () -> {
            new BinDec_8bits("0000111X"); // Invalid character
        });
        assertEquals("La cadena binaria solo puede contener '0' y
'1'", thrown.getMessage());
    }

    @Test
    public void testMaxValue() {
        BinDec_8bits binDec = new BinDec_8bits("11111111"); // Maximum
8-bit value
        assertEquals(255, binDec.BinADecimal());
    }

    @Test
```

```
    public void testMinValue() {
        BinDec_8bits binDec = new BinDec_8bits("00000000"); // Minimum
8-bit value
        assertEquals(0, binDec.BinADecimal());
    }
}
```

Pruebas para decimalBinario8bits

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class Dec_bin8bitsTest {

    @Test
    public void testDecimalABin() {
        dec_bin8bits decBin = new dec_bin8bits(15);
        assertEquals("00001111", decBin.DecimalABin());
    }

    @Test
    public void testMaxValue() {
        dec_bin8bits decBin = new dec_bin8bits(255); // Maximum 8-bit
value
        assertEquals("11111111", decBin.DecimalABin());
    }

    @Test
    public void testMinValue() {
        dec_bin8bits decBin = new dec_bin8bits(0); // Minimum 8-bit
value
        assertEquals("00000000", decBin.DecimalABin());
    }

    @Test
    public void testInvalidNegativeValue() {
        IllegalArgumentException thrown =
assertThrows(IllegalArgumentException.class, () -> {
            new dec_bin8bits(-1); // Invalid negative value
        });
        assertEquals("El número decimal debe estar en el rango de 0 a
255", thrown.getMessage());
    }

    @Test
    public void testInvalidOverMaxValue() {
        IllegalArgumentException thrown =
assertThrows(IllegalArgumentException.class, () -> {
            new dec_bin8bits(256); // Value exceeds 8-bit range
        });
        assertEquals("El número decimal debe estar en el rango de 0 a
255", thrown.getMessage());
    }
}
```

Pruebas para ExpresionATrabajar

```
package calculadoraBooleana;

import static org.junit.jupiter.api.Assertions.*;
```

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class ExpresionATrabajarTest {

    private ExpresionATrabajar exp;

    @BeforeEach
    public void setUp() {
        exp = new ExpresionATrabajar();
    }

    @Test
    public void testExprTrabValidExpression() {
        String expression = "A+B*(~C)";
        assertDoesNotThrow(() -> exp.exprTrab(expression));
    }

    @Test
    public void testExprTrabInvalidExpression() {
        String expression = "A+B*D";
        Exception exception =
assertThrows(IllegalArgumentException.class, () ->
exp.exprTrab(expression));
        assertEquals("Expresión inválida", exception.getMessage());
    }

    @Test
    public void testAsoMult() {
        OperacionesBooleanas op = new OperacionesBooleanas();
        String expression = "(A+B)+C";
        String result = exp.asoMult(expression, op);
        assertNotNull(result);
        // Agregar más aserciones según la lógica de asociación de
multiplicaciones.
    }

    @Test
    public void testAsoSums() {
        OperacionesBooleanas op = new OperacionesBooleanas();
        String expression = "(A+B)+C";
        String result = exp.asoSums(expression, op);
        assertNotNull(result);
        // Agregar más aserciones según la lógica de asociación de
sumas.
    }
}
```

Pruebas para ExpressionParser

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class ExpressionParserTest {

    @Test
    void testParseValidExpression() {
        ExpressionParser parser = new ExpressionParser();
        Node result = parser.parse("true AND false");
        assertNotNull(result);
        assertEquals("AND", result.value);
```

```
        assertEquals("true", result.left.value);
        assertEquals("false", result.right.value);
    }

    @Test
    void testParseNullExpression() {
        ExpressionParser parser = new ExpressionParser();
        Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
            parser.parse(null);
        });
        assertEquals("La expresión no puede ser nula o vacía.",
exception.getMessage());
    }

    @Test
    void testParseEmptyExpression() {
        ExpressionParser parser = new ExpressionParser();
        Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
            parser.parse("");
        });
        assertEquals("La expresión no puede ser nula o vacía.",
exception.getMessage());
    }

    @Test
    void testParseInvalidOperand() {
        ExpressionParser parser = new ExpressionParser();
        Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
            parser.parse("true AND invalidOperand");
        });
        assertEquals("Operando no válido: invalidOperand",
exception.getMessage());
    }

    @Test
    void testParseUnbalancedParentheses() {
        ExpressionParser parser = new ExpressionParser();
        Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
            parser.parse("( true AND false");
        });
        assertEquals("Paréntesis desbalanceados en la expresión.",
exception.getMessage());
    }
}
```

Pruebas para ExpressionSimplifier

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class ExpressionSimplifierTest {

    @Test
    void testSimplifyValidExpression() {
        ExpressionSimplifier simplifier = new ExpressionSimplifier();
        Node node = new Node("AND", new Node("true"), new
Node("false"));
```

```
        Node result = simplifier.simplify(node);
        assertEquals("false", result.value);
    }

    @Test
    void testSimplifyInvalidNode() {
        ExpressionSimplifier simplifier = new ExpressionSimplifier();
        Node node = new Node("INVALID", new Node("true"), new
Node("false"));
        Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
            simplifier.simplify(node);
        });
        assertEquals("Nodo no válido: INVALID",
exception.getMessage());
    }
}
```

Pruebas para Node

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class NodeTest {

    @Test
    void testValidNode() {
        Node node = new Node("true");
        assertEquals("true", node.value);
    }

    @Test
    void testInvalidNode() {
        Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
            new Node("INVALID");
        });
        assertEquals("Valor no válido para el nodo: INVALID",
exception.getMessage());
    }
}
```

Pruebas para OperacionesBooleanas

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class OperacionesBooleanasTest {

    @Test
    void testEvaluarValidExpression() {
        Node node = new Node("AND", new Node("true"), new
Node("false"));
        assertFalse(OperacionesBooleanas.evaluar(node));
    }

    @Test
    void testEvaluarNullNode() {
        Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
```

```
            OperacionesBooleanas.evaluar(null);
        });
        assertEquals("El nodo no puede ser nulo.",
exception.getMessage());
    }

    @Test
    void testEvaluarInvalidNode() {
        Node node = new Node("INVALID", new Node("true"), new
Node("false"));
        Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
            OperacionesBooleanas.evaluar(node);
        });
        assertEquals("Nodo no válido: INVALID",
exception.getMessage());
    }
}
```

7. Mejoras y Recomendaciones

En caso de modificación del código, se recomienda seguir las siguientes cualidades:

- **Verificación Exhaustiva de Entradas:**
  - Siempre verifique las entradas del usuario para asegurarse de que estén en el formato y rango esperados. Esto puede incluir la longitud de la cadena, el rango de valores numéricos y los caracteres permitidos.
- **Incorporar Manejo de Excepciones:**
  - Manejar excepciones de manera efectiva para proporcionar mensajes de error claros y específicos que ayuden al usuario a corregir su entrada.
- **Pruebas Unitarias Extensas:**
  - Asegúrese de incluir pruebas unitarias para todas las posibles entradas, incluidas las válidas y las inválidas, para garantizar que su código se comporte como se espera en todos los casos.
- **Optimización de Código:**
  - Optimice el código para mejorar la legibilidad y el rendimiento. Por ejemplo, utilice métodos integrados de conversión en lugar de implementar manualmente algoritmos de conversión.
- **Documentación Clara:**
  - Documente claramente cada clase y método para explicar su propósito, parámetros, valores de retorno y excepciones lanzadas.

Importante: No modifique el código si no está seguro de su funcionamiento. En caso de dudas, se recomienda contactar con el equipo de desarrollo:

BackEnd	**Contacto**
Mark Hernández	Email: marck.hernandez@epn.edu.ec Teléfono: 0987880753
Anthony Contreras	Email: anthony.contreras@epn.edu.ec Teléfono: 0962879095
FrontEnd	**Contacto**
Yasid Jiménez	Email: yasid.jimenez@epn.edu.ec Teléfono: 0999073691
Edison Quizhpe	Email: edison.quizhpe@epn.edu.ec Teléfono: 0961053724