Checklist de Proyecto

Verificación general del código

Estructura y Organización

1. **Estructura del archivo**
   - Los archivos tienen un nombre significativo y adecuado.
   - El nombre de la clase coincide con el nombre del archivo.
   - La clase contiene un método `main o principal` que inicia el programa.
2. **Paquetes e imports**
   - Se utilizan los paquetes necesarios (`java.util.Scanner` en este caso).
   - No hay imports innecesarios.

Funcionalidad

3. **Entrada y Salida**
   - El programa solicita la entrada adecuada del usuario.
   - La entrada del usuario se lee correctamente.
   - El programa convierte correctamente los valores (binario a decimal, binario a hexadecimal, decimal a binario, hexadecimal a binario).
   - La salida es clara y muestra el resultado esperado.

Buenas Prácticas

4. **Comentarios y Documentación**
   - El código tiene comentarios explicativos donde sea necesario.
   - El propósito del programa y de las funciones está documentado.
5. **Manejo de Errores**
   - Se manejan adecuadamente las posibles excepciones (ej. entrada no válida, rangos no validos).
6. **Cierre de Recursos**
   - Los recursos como `Scanner` se cierran después de su uso.

Estilo de Código

7. **Formato y Legibilidad**
   - El código está correctamente indentado.
   - Los nombres de variables son significativos y siguen las convenciones de nomenclatura (camelCase para variables y métodos).
   - No hay líneas de código innecesarias o comentadas sin razón.
8. **Eficiencia**
   - El código es eficiente, ordenado y no contiene redundancias (bajo coste).
   - El código no presenta datos quemados.

Verificación estructural y funcionalidad (clase por clase)

Revisión de binarioDecimal

- **Separación de bits en signo, exponente y mantisa**:
  - Método `separacion()` separa correctamente el bit de signo (`ssgn`), los bits del exponente (`sexp`), y la mantisa (`smant`).
- **Cálculo del exponente y valor decimal**:
  - Método `exponente()` calcula el exponente ajustando por el sesgo de 127.
  - Método `numDec()` maneja los casos especiales y utiliza `bin_Dec_D` para calcular la mantisa.
- **Manejo de casos especiales (cero, infinito, NaN)**:
  - `numDec()` maneja correctamente los casos para exponentes de todo ceros y todo unos.

Revisión binarioHexadecimal

- **Verificación de la longitud del binario**:
  - Método `verificar()` asegura que la longitud del binario sea de 32 bits, añadiendo ceros al inicio si es necesario.
- **Conversión de grupos de 4 bits a caracteres hexadecimales**:
  - Método `hex()` convierte cada grupo de 4 bits a su correspondiente carácter hexadecimal mediante un `switch`.

Revisión de decimalBinario

- **Conversión de la parte entera y la parte decimal a binario**:
  - Métodos `entera()` y `decimal()` realizan las conversiones respectivas correctamente.
- **Ensamblaje del formato IEEE 754**:
  - Método `ieee()` construye el formato IEEE 754 correctamente, manejando el signo, exponente y mantisa.

Revisión de hexadecimalBinario

- **Verificación de la longitud del hexadecimal**:

- Método `verificar()` asegura que la longitud del hexadecimal sea de 8 caracteres, añadiendo ceros al inicio si es necesario.

- **Conversión de cada carácter hexadecimal a 4 bits binarios**:

  - Método `bin()` convierte cada carácter hexadecimal a su correspondiente representación binaria de 4 bits mediante un `switch`.

Revisión de binarioDecimal8bits

- **Verificación de la Longitud del binario**:

  - El constructor de la clase asegura que la longitud del binario (`nBin`) sea de 8 caracteres. Si la longitud es menor a 8, añade ceros al inicio hasta completar 8 caracteres.

- **Conversión de Carácter Hexadecimal a Binario**:

  - El método `transformar()` analiza si el primer bit es 0 o 1, y ajusta el valor decimal en consecuencia.

Revisión de decimalBinario8bits

Conversión de Carácter decimal a Binario:

- El método `binario()` convierte un número entero a su representación binaria de 8 bits, asegurando que siempre tenga 8 caracteres añadiendo ceros al inicio si es necesario.

ExpresionATrabajar.java

Verificación Estructural

1. **Paquete**:
   - El archivo pertenece al paquete `calculadoraBooleana`.
2. **Definición de Clase**:
   - La clase `ExpresionATrabajar` está correctamente definida y es pública.
3. **Atributos**:
   - `String medio = "salida 1";`: Atributo de clase para almacenar el estado intermedio.
4. **Métodos**:
   - `exprTrab(String expression)`: Método principal para simplificar la expresión booleana.
   - `asoMult(String expression, OperacionesBooleanas op)`: Método para asociar multiplicaciones en la expresión.
   - `asoSums(String expression, OperacionesBooleanas op)`: Método para asociar sumas en la expresión.
5. **Comentarios**:

- Los comentarios proporcionan ejemplos de expresiones válidas y no válidas y explican el propósito de los métodos.

Verificación de Funcionalidad

1. **Método `exprTrab(String expression)`:**
   - **Propósito**: Simplificar la expresión booleana proporcionada.
   - **Proceso**:
     - Se utiliza `ExpressionParser` para analizar la expresión y convertirla en un árbol de nodos.
     - Se simplifica la expresión usando `ExpressionSimplifier`.
     - Se ajusta la expresión si está entre paréntesis.
     - Se aplica la asociación de multiplicaciones y sumas usando `asoMult` y `asoSums`.
     - El proceso se repite hasta que la expresión no cambie más.
   - **Manejo de Excepciones**: No se maneja explícitamente en este método, se asume que las clases utilizadas (`ExpressionParser`, `ExpressionSimplifier`, `OperacionesBooleanas`) manejan sus propias excepciones.
2. **Método `asoMult(String expression, OperacionesBooleanas op)`:**
   - **Propósito**: Asociar multiplicaciones en la expresión.
   - **Proceso**:
     - Utiliza expresiones regulares para encontrar patrones de multiplicación en la expresión.
     - Usa el método `asociarMultiplicaciones` de `OperacionesBooleanas` para procesar y ajustar la expresión.
   - **Manejo de Excepciones**: No se maneja explícitamente.
3. **Método `asoSums(String expression, OperacionesBooleanas op)`:**
   - **Propósito**: Asociar sumas en la expresión.
   - **Proceso**:
     - Utiliza expresiones regulares para encontrar patrones de suma en la expresión.
     - Usa el método `asociarSumas` de `OperacionesBooleanas` para procesar y ajustar la expresión.
   - **Manejo de Excepciones**: No se maneja explícitamente.

ExpressionParser.java

Verificación Estructural

1. **Imports y Paquete**:
   - Asegúrate de importar las clases necesarias (`HashMap`, `Stack`).
2. **Definición de Clase**:
   - La clase `ExpressionParser` debe estar correctamente definida y pública.
3. **Constructores y Métodos**:
   - Constructor para inicializar los operadores.
   - Método `parse(String expression)` para analizar la expresión.
   -

Verificación de Funcionalidad

1. **Inicialización de Operadores**:
    - Los operadores `OR`, `AND`, `NOT` deben estar correctamente inicializados en el `HashMap` con sus respectivas prioridades.
2. **Análisis de Expresiones**:
    - El método `parse` debe dividir la expresión en tokens y utilizar pilas para construir el árbol de nodos.
    - Validaciones de expresión nula, vacía y paréntesis desbalanceados.
3. **Creación de Nodos**:
    - Crear nodos para operadores y operandos.
    - Manejo de precedencia de operadores usando pilas.
4. **Manejo de Excepciones**:
    - Lanza `IllegalArgumentException` para expresiones nulas, vacías, operandos no válidos y paréntesis desbalanceados.

ExpressionSimplifier.java

Verificación Estructural

1. **Imports y Paquete**:
    - Verifica que no haya imports innecesarios.
2. **Definición de Clase**:
    - La clase `ExpressionSimplifier` debe estar correctamente definida y pública.
3. **Métodos**:
    - Método `simplify(Node node)` para simplificar la expresión.

Verificación de Funcionalidad

1. **Simplificación de Nodos**:
    - El método `simplify` debe manejar correctamente los operadores `AND`, `OR`, y `NOT`.
    - Debe retornar el valor simplificado del nodo.
2. **Manejo de Excepciones**:
    - Lanza `IllegalArgumentException` si el nodo contiene un valor no válido.

Node.java

Verificación Estructural

1. **Definición de Clase**:
    - La clase `Node` debe estar correctamente definida y pública.
2. **Atributos**:
    - Atributos `value`, `left`, `right` deben estar definidos como públicos.
3. **Constructores**:
    - Constructor `Node(String item)` para nodos simples.

- Constructor `Node(String value, Node left, Node right)` para nodos compuestos.

Verificación de Funcionalidad

1. **Inicialización de Nodos**:
   - Los constructores deben inicializar correctamente los atributos del nodo.
   - Validaciones de valores de nodo en los constructores.
2. **Manejo de Excepciones**:
   - Lanza `IllegalArgumentException` si el valor del nodo es no válido.

OperacionesBooleanas.java

Verificación Estructural

1. **Definición de Clase**:
   - La clase `OperacionesBooleanas` debe estar correctamente definida y pública.
2. **Métodos**:
   - Método `evaluar(Node node)` para evaluar la expresión booleana.

Verificación de Funcionalidad

1. **Evaluación de Nodos**:
   - El método `evaluar` debe evaluar correctamente los operadores `AND`, `OR`, `NOT`.
   - Retornar el valor booleano de la evaluación.
2. **Manejo de Excepciones**:
   - Lanza `IllegalArgumentException` si el nodo es nulo.
   - Lanza `IllegalArgumentException` si el nodo contiene un valor no válido.

Ejecución del programa

Interfaz

- El programa cuenta con una interfaz limpia, completa y amigable con el usuario.
- El programa cuenta con validaciones que controlan el tipo de información que se ingresa.
- El programa no inicia desde la pantalla de bienvenida. …
- En ocasiones, el programa deja de funcionar o directamente no funciona al activar el tipo de formato (IEEE). …
- En la mayoría de los casos, el programa no indica limites en cuanto al ingreso de datos (se detalla en pruebas unitarias). …
- En ocasiones, el historial no muestra el resultado obtenido. …

Pruebas unitarias (Conversiones, limites, desbordamientos y casos especiales)

Se ejecutan pruebas en las distintas funciones del programa.

Decimal a binario

1. **Conversiones**
    - En casos ideales, el programa funciona correctamente, transformando el numero decimal a binario de 8 bits o en formato IEEE.
2. **Limites, desbordamiento y casos especiales**
    - El programa funciona correctamente sin importar si se ingresan números muy grandes o negativos.

Binario a decimal

1. **Conversiones**
    - En casos ideales, el programa funciona correctamente, tanto en 8 bits como en formato IEEE.
2. **Límites y desbordamiento**
    - El programa completa de ceros en caso de que así lo requiera la conversión.
    - No existe un control sobre valores ingresados mayores a 8 bits y 32 bits.
3. **Casos especiales**
    - El programa cuenta con funcionalidad en los siguientes casos:
        - Muestra como resultado el 0.
        - Muestra NaN si no es posible la conversión.
        - Muestra mas o menos infinito como resultado en ciertos casos.

Hexadecimal a binario

- **Observación:** El botón “borrar historial” no funciona en esta función. …

1. **Conversiones**
    - El programa sin la opción de formato IEEE no muestra el resultado en el cuadro de texto, no calcula o muestra un resultado erróneo. …
    - Para formato IEEE, y en casos ideales, el programa funciona correctamente.
2. **Límites y desbordamiento**
    - El programa no cuenta con validación de datos erróneos (Ej. 1Z). …

Binario a Hexadecimal

- **Observación:** El cuadro de historial muestra información errónea. …

1. **Conversiones:**
    - El programa sin la opción de formato IEEE directamente no funciona. …
    - Para formato IEEE, y en casos ideales, el programa funciona correctamente.
    - **Posible corrección:** En formato IEEE, la respuesta se autocompleta de ceros (Ej. 0004C). …
2. **Límites y desbordamiento**
    - El programa funciona, aunque se ingrese un numero binario menor a 32 bits.

- El programa tiene control para restringir el ingreso de datos no validos (Datos diferentes a 1 y 0).

Conclusiones

Estructura y Organización

- **Buena estructura:** Los archivos están bien nombrados y las clases están correctamente definidas con un método `main` que sirve como punto de entrada.
- **Uso adecuado de paquetes:** Los paquetes necesarios están importados correctamente, y no hay imports innecesarios, lo cual es una buena práctica.

Buenas Prácticas

- **Comentarios y documentación:** El código se encuentra comentado debidamente, facilitando la legibilidad y reestructuración.

Estilo de Código

- **Formato y legibilidad:** El código está bien formateado y es legible, siguiendo convenciones de nomenclatura estándar.
- **Eficiencia:** El código es eficiente y no tiene redundancias significativas.

Identificación de Errores

- **Entrada inválida:** En varias funciones del programa hace falta implementar limites de entrada de datos. …

Funcionalidad

- **Funcionalidad básica:** Los programas leen entradas del usuario y realizan las conversiones de manera adecuada, a excepción de las funciones `binarioHexadecimal` y `hexadecimalBinario`, donde la funcionalidad se ve severamente afectada, llegando incluso a no funcionar. Se requiere una nueva revisión del código y posible reestructuración. …
- **Mensajes claros:** Los mensajes que solicitan entrada del usuario son claros y directos, facilitando la interacción.

Corrección de errores (Versión Final)

1. Se soluciono el problema por el cual el programa no se ejecutaba desde la ventana de bienvenida
2. Se soluciono el problema por el cual el programa dejaba de funcionar al activar la opción de Formato IEEE 754
3. Se corrigió el problema que no permitía al historial mostrar la información guardada
4. En binario a decimal, se agregó validaciones para permitir solo el ingreso de valores en 8 o 32 bits.
5. Se arreglo el botón “borrar historial” en hexadecimal a binario.

6. Se soluciono el problema por el cual, en hexadecimal a binario, el programa no calculaba sin la opción de Formato IEEE 754
7. Se agrego mas validaciones a la entrada de hexadecimal a binario.
8. Se arreglo el problema por el cual el historial mostraba información errónea en binario a hexadecimal.
9. Se arreglo el problema por el cual el programa no funcionaba sin la opción de Formato IEEE 754 en binario a hexadecimal.
10. Se cambio la forma de salida en binario a hexadecimal, mostrando ahora si el resultado correcto.

Conclusiones Finales

Estructura y Organización

- **Buena estructura:** Los archivos están bien nombrados y las clases están correctamente definidas con un método `main` que sirve como punto de entrada.
- **Uso adecuado de paquetes:** Los paquetes necesarios están importados correctamente, y no hay imports innecesarios, lo cual es una buena práctica.

Buenas Prácticas

- **Comentarios y documentación:** El código se encuentra comentado debidamente, facilitando la legibilidad y reestructuración.

Estilo de Código

- **Formato y legibilidad:** El código está bien formateado y es legible, siguiendo convenciones de nomenclatura estándar.
- **Eficiencia:** El código es eficiente y no tiene redundancias significativas.

Identificación de Errores

- **Entrada inválida:** Se ha implementado el máximo de validaciones posibles para garantizar que no exista errores.

Funcionalidad

- **Funcionalidad básica:** Los programas leen entradas del usuario y realizan las conversiones de manera adecuada, así mismo, calculan de forma correcta las operaciones booleanas.
- **Mensajes claros:** Los mensajes que solicitan entrada del usuario son claros y directos, facilitando la interacción.