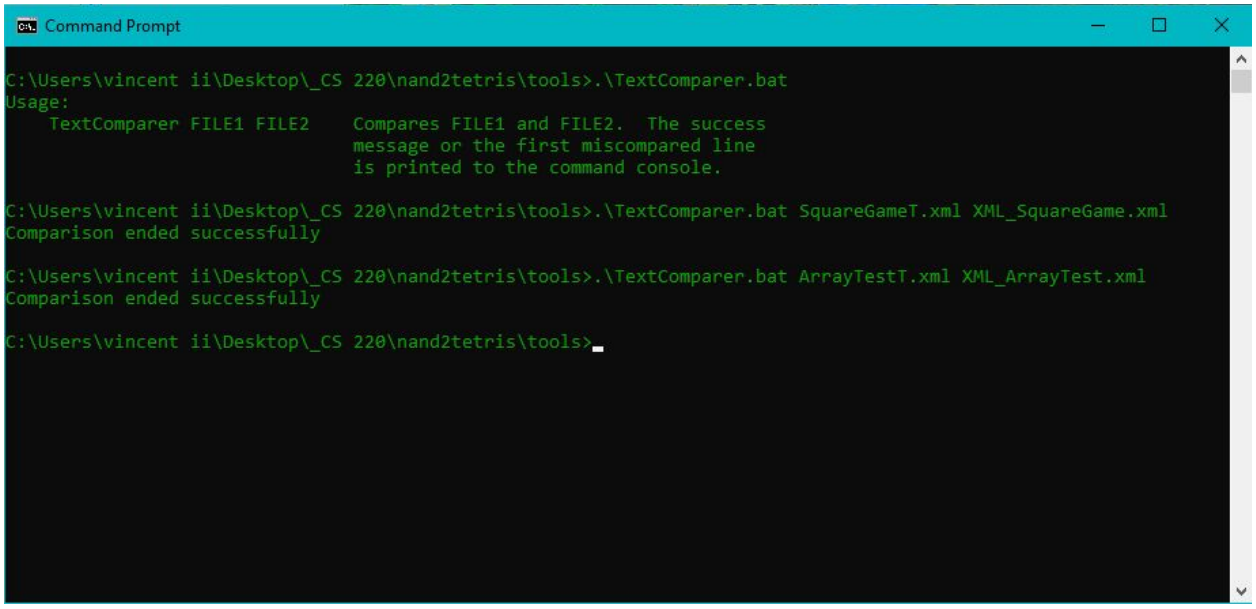
Lab 11 HW

SquareGameT.xml comparison & ArrayTestT.xml comparison



Code for program:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Hashtable;
import java.util.Scanner;
import java.util.StringTokenizer;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * This class handles parsing a jack file, and writing the tokens parsed to an XML file.
 *
 * Algorithm:
 * 1| Constructor builds out the printWriter & scanner utils.
 * 2| Creates a tokenizer from the fileName.
 * 3| Writes the official token starting tag onto the XML file.
 * 4| Advances through entire open file using the writer until no more lines can be translated.
 * 5| writes the closing tag for tokens.
 * 6| closes the printWriter and finishes comp successfully.
 *
```

```
 * @author vincentii
 * @version 1.0
 */
public class JackTokenizer {

    // instanced variables
    private static char symbol;
    private static int intVal;
    private static int tokenCount;
    private static String cleanInput;
    private static String stringVal;
    private static Keyword keyword;
    private static TokenType tokenType;
    private static Scanner inputFile;
    private static PrintWriter writer;
    private static Hashtable<Character, TokenType> symbolTable;
    private static Hashtable<String, Keyword> keywordTable;

    /**
     * Main Method, handles the driving of the program.
     * @param args ignored.
     */
    public static void main(String[] args) {
        // gathers input of fileName from the user.
        inputFile = new Scanner(System.in);
        System.out.print("Please enter a valid jack file to tokenize: ");
        String fileName = inputFile.nextLine();
        inputFile.close();

        // creates tokenizer
        JackTokenizer tokenizer = new JackTokenizer(fileName);

        // writes tokens tag
        writer.write("<tokens>\n");

        // parses the file
        while (tokenizer.hasMoreTokens()) {
            tokenizer.advance(writer);
        }

        // closes tokens tag
        writer.write("</tokens>");
```

```
    // closes writer
    writer.close();

    // lets user known the compilation worked.
    System.out.println("Comp was successful.");
}

/**
 * handles parsing 1 line from the jack file.
 * Algorithm:
 * 1 - Davide cleanLine into many tokens.
 * 2 - Davide tokens into even smaller subTokens if they have tokens adjacent to each other.
 * 3 - Classify the token or subToken, and write to XML file.
 *
 * pre: the writer is open and valid, the file is in good jack format ready to be compiled.
 * post: 1 cleanLine from the jack file is translated into many types of tokens and written
 *       to an XML file.
 * @author vincent ii
 * @version 1.0
 */
public void advance(PrintWriter writer) {
    String current_line = inputFile.nextLine();

    // System.out.println("Current Line: " + current_line + "  |  ");

    // gets rid of block comments
    if  (current_line.contains("/**") || current_line.contains("*/") ||
        current_line.contains("*")) {
        current_line = "";
    }

    // gets rid of comments
    if (current_line.contains("//")) {
        if (current_line.charAt(0) == '/') {
            current_line = "";
        } else {
            String[] strings = current_line.split("//");
            current_line = strings[0];
        }

    }

    // trims
```

```
        cleanInput = current_line.trim();

        // System.out.println("Parsing line: " + cleanInput);

        // ignores empty input (comments & blank lines)
        if (cleanInput.isEmpty()) {
            return;
        }

        // tokenize the line.
        StringTokenizer tokenizer = new StringTokenizer(cleanInput);

        // System.out.println("Clean Line: " + cleanInput);

        // runs through the tokens.
        while (tokenizer.hasMoreTokens()) {
            // classify the token
            String token = tokenizer.nextToken();
            // System.out.println("\tCurrent token: " + token);

            // last symbol
            int lastSymbol = 0;

            // loops through chars in current token, looking to further cleanse the String from any
symbols.
            for (int i = 0; i < token.length(); i++) {

                // System.out.println(token.substring(lastSymbol, i+1));

                // checks for string constant.
                if (token.charAt(i) == '"') {
                    // makes sure the quote isn't next to a symbol.
                    if (!symbolTable.containsKey(token.charAt(i + 1))) {
                        // System.out.printf("\t\tString Constant detected: ");

                        // creates a string between the " "
                        Pattern quotes = Pattern.compile("\"([^\"]*)\"");
                        Matcher matcher = quotes.matcher(cleanInput);
                        matcher.find();
                        stringVal = matcher.group(0);
                        stringVal = stringVal.substring(1);
                        stringVal = stringVal.substring(0, stringVal.indexOf('"'));
                        tokenType = TokenType.STRING_CONST;
```

```
                // writes the token label in XML
                printElement("stringConstant", stringVal, writer);
                // System.out.printf(tokenCount + "| " + tokenType + ", " + stringVal + "\n");

                // increments the index of the token to the end of the string constant.
                tokenCount++;
                StringTokenizer anotherTokenizer = new StringTokenizer(stringVal);

                // gets rid of unneeded tokens from cleanLine tokenizer.
                for (int j = 0; j < anotherTokenizer.countTokens() - 1; j++) {
                    if (tokenizer.hasMoreTokens()) {
                        // System.out.println(tokenizer.nextToken());
                        tokenizer.nextToken();
                    }
                }

                continue;
            } else {
                token = token.substring(1);
                // System.out.println(token);
            }

        }

        // checks for a keyWord, starting from the lastSymbol to the current character.
        if (keywordTable.containsKey(token.substring(lastSymbol, i+1))) {
            // System.out.printf("\t\tKeyword detected: ");

            // gathers the keyword
            keyword = keywordTable.get(token.substring(lastSymbol, i+1));
            tokenType = TokenType.KEYWORD;

            // writes the token label in XML
            printElement("keyword", keyword.toString(), writer);
            // System.out.printf(tokenCount + "| " + tokenType + ", " + keyword.toString() + "\n");

            // increments the index of the token to the end of the string constant.
            tokenCount++;
            lastSymbol = i + 1; // moves the lastSymbol because we have extracted a keyword
from the token.
            continue;
        }
```

```
            // checks for the token being an identifier.
            if (!symbolTable.containsKey(token.charAt(lastSymbol)) &&
                  !Character.isDigit(token.charAt(lastSymbol))) {

               // cI+1 must be a symbol, or last char of token.
               if (i + 1 == token.length()) {
                  // System.out.printf("\t\tIdentifier detected: ");

                  // subString from lastSym to i must be a identifier.
                  stringVal = token.substring(lastSymbol);
                  tokenType = TokenType.IDENTIFIER;

                  // writes the token label in XML
                  printElement("identifier", stringVal, writer);
                  // System.out.printf(tokenCount + "| " + tokenType + ", " + stringVal + "\n");

                  // increments the index of the token to the end of the string constant.
                  tokenCount++;
                  continue;

               } else if (symbolTable.containsKey(token.charAt(i + 1))) {
                  // System.out.printf("\t\tIdentifier detected: ");

                  // subString from lastSym to i must be a identifier.
                  stringVal = token.substring(lastSymbol, i + 1);
                  tokenType = TokenType.IDENTIFIER;

                  // writes the token label in XML
                  printElement("identifier", stringVal, writer);
                  // System.out.printf(tokenCount + "| " + tokenType + ", " + stringVal + "
symNext\n");

                  // increments the index of the token to the end of the string constant.
                  tokenCount++;
                  lastSymbol = i + 1; // moves the lastSymbol because we have extracted a
keyword from the token.
                  continue;
               }

            }
```

```
// checks for a int constant.
if (Character.isDigit(token.charAt(i))) {

    // checks if this is a larger number.
    if (Character.isDigit(token.charAt(i + 1))) {
        // can continue, larger than 1 digit number.
        continue;
    } else {
        // System.out.printf("\t\tInteger Constant detected: ");

        // must write because next char is not a int.
        // handles 1 digit place and larger numbers
        if ((i - lastSymbol) == 0) {
            intVal = Character.getNumericValue(token.charAt(i));
        } else {
            String tempValue = token.substring(lastSymbol, i+1);
            // System.out.println(tempValue);
            intVal = Integer.valueOf(tempValue);
        }
        tokenType = TokenType.INT_CONST;

        // writes the token label in XML
        printElement("integerConstant", String.valueOf(intVal), writer);
        // System.out.printf(tokenCount + "| " + tokenType + ", " + intVal + "\n");

        tokenCount++;
        lastSymbol = i; // moves the lastSymbol detected variable.
        continue;
    }

}


// checks for symbol
if (symbolTable.containsKey(token.charAt(i))) {
    // System.out.printf("\t\tSymbol detected: ");

    // gathers the symbol
    symbol = token.charAt(i);
    tokenType = TokenType.SYMBOL;

    // writes the token label in XML
```

```
                printElement("symbol", String.valueOf(symbol), writer);
                // System.out.printf(tokenCount + "| " + tokenType + ", " + symbol + " indexFound: "
+ i + "\n");

                // increments the index of the token to the end of the string constant.
                tokenCount++;
                lastSymbol = i + 1; // moves the lastSymbol detected variable.
                continue;
            }


        }

    }

}

/**
 * Constructor for a new jackTokenizer object. Will setup the fileWriter.
 * pre: fileName is valid, path is valid.
 * post: Will create a new file with XML_ appended to the front, and read from the fileName.
 * @param fileName the name of the file to build an XML from, and read from.
 */
public JackTokenizer(String fileName) {
    // sets up reading and writing.
    try {
        writer = new PrintWriter(new File(("XML_" + fileName.substring(0, fileName.indexOf('.'))
+ ".xml")));
        inputFile = new Scanner(new File(fileName));
    } catch (FileNotFoundException e) {
        System.out.println("Problem reading from file " + fileName + " exiting program.");
        System.exit(0);
    }
    System.out.println(fileName + " is ready to be parsed!");

    // builds symbolTable
    symbolTable = new Hashtable<>();
    symbolTable.put('{', TokenType.SYMBOL);
    symbolTable.put('}', TokenType.SYMBOL);
    symbolTable.put('(', TokenType.SYMBOL);
    symbolTable.put(')', TokenType.SYMBOL);
    symbolTable.put('[', TokenType.SYMBOL);
    symbolTable.put(']', TokenType.SYMBOL);
```

```
symbolTable.put('.', TokenType.SYMBOL);
symbolTable.put(',', TokenType.SYMBOL);
symbolTable.put(';', TokenType.SYMBOL);
symbolTable.put('+', TokenType.SYMBOL);
symbolTable.put('-', TokenType.SYMBOL);
symbolTable.put('*', TokenType.SYMBOL);
symbolTable.put('/', TokenType.SYMBOL);
symbolTable.put('&', TokenType.SYMBOL);
symbolTable.put('|', TokenType.SYMBOL);
symbolTable.put('<', TokenType.SYMBOL);
symbolTable.put('>', TokenType.SYMBOL);
symbolTable.put('=', TokenType.SYMBOL);
symbolTable.put('~', TokenType.SYMBOL);
symbolTable.put('"', TokenType.STRING_CONST);
System.out.println("symbolTable has been built successfully!");

// builds keywordTable
keywordTable = new Hashtable<>();
keywordTable.put("class", Keyword.CLASS);
keywordTable.put("constructor", Keyword.CONSTRUCTOR);
keywordTable.put("function", Keyword.FUNCTION);
keywordTable.put("method", Keyword.METHOD);
keywordTable.put("field", Keyword.FIELD);
keywordTable.put("static", Keyword.STATIC);
keywordTable.put("var", Keyword.VAR);
keywordTable.put("int", Keyword.INT);
keywordTable.put("char", Keyword.CHAR);
keywordTable.put("boolean", Keyword.BOOLEAN);
keywordTable.put("void", Keyword.VOID);
keywordTable.put("true", Keyword.TRUE);
keywordTable.put("false", Keyword.FALSE);
keywordTable.put("null", Keyword.NULL);
keywordTable.put("this", Keyword.THIS);
keywordTable.put("that", Keyword.THAT);
keywordTable.put("let", Keyword.LET);
keywordTable.put("do", Keyword.DO);
keywordTable.put("if", Keyword.IF);
keywordTable.put("else", Keyword.ELSE);
keywordTable.put("while", Keyword.WHILE);
keywordTable.put("return", Keyword.RETURN);
System.out.println("keywordTable has been built successfully!");

tokenCount = 0;
```

```
    }


    /**
     * Handles writing entire tag line to a XML file using a printWriter.
     * pre: Valid tag, value and open writer.
     * post: 1 line of text containing all of the above in XML format.
     * @param tag the tag to write should be a keyword, or a identifier.
     * @param value the value to write, what goes inside of the tag.
     * @param writer the fileWriter to use to actually perform the write.
     */
    public void printElement(String tag, String value, PrintWriter writer) {
        // checks if value is a bad XML value and needs to be substituted.
        switch (value) {
            case ("&"):
                value = "&amp;";
                break;
            case ("<"):
                value = "&lt;";
                break;
            case (">"):
                value = "&gt;";
                break;
            default:
                break;
        }
        writer.write("<" + tag + ">");
        writer.write(" " + value + " ");
        writer.write("</" + tag + ">\n");
        writer.flush();
    }

    /**
     * pre: A valid tokenizer exists, with a valid inputFile Scanner object.
     * post: If there are any more lines to parse from the inputFile object.
     * @return true if more lines need to be parsed, false if end of file.
     */
    public boolean hasMoreTokens() {
        return inputFile.hasNextLine();
    }

}
```

Enums:

```
/**
 * Handles storing the type of keyword.
 * @author vincentii
 * @version 1.0
 */
public enum Keyword {
    CLASS,
    METHOD,
    FUNCTION,
    CONSTRUCTOR,
    INT,
    BOOLEAN,
    CHAR,
    VOID,
    VAR,
    STATIC,
    FIELD,
    LET,
    DO,
    IF,
    ELSE,
    WHILE,
    RETURN,
    TRUE,
    FALSE,
    NULL,
    THAT,
    THIS;

    @Override
    public String toString() {
        return this.name().toLowerCase();
    }

}
```

```
/**
 * Handles the type of the token.
 * @author vincentii
 * @version 1.0
 */
public enum TokenType {
    KEYWORD,
    SYMBOL,
    INT_CONST,
    STRING_CONST,
    IDENTIFIER;

    @Override
    public String toString() {
        return this.name().toLowerCase();
    }

}
```