Name: Victor Hazali

Tutorial Group ID: W09

CE2 Code

TextBuddy.java

```
package cs2103;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;

/**
 * This class is used to manipulate a text file. It supports 6 basic commands:
 * add, delete, display, clear, save and exit
 *
 * @author Victor Hazali
 */
public class TextBuddy {

	private static final String	WELCOME_MESSAGE		= "Welcome to TextBuddy. %1$s is ready for use";
	private static final String	MESSAGE_FILE_DELETED	= "Deleted from %1$s: \"%2$s\"";
	private static final String	MESSAGE_ALL_DELETED	= "All content deleted from %1$s";
	private static final String	MESSAGE_FILE_ADDED	= "Added to %1$s: \"%2$s\"";
	private static final String	MESSAGE_FILE_EMPTY	= "%1$s is empty";
	private static final String	MESSAGE_INVALID_COMMAND		= "%1$s is an invalid command format";
	private static final String	MESSAGE_FILE_IO_ERROR	= "Failed to open %1$s";
	private static final String	MESSAGE_FILE_SAVED	= "%1$s successfully saved";
	private static final String	MESSAGE_REPEATED_INPUT		= "%1s is already added";
	private static final String	MESSAGE_WRONG_INDEX	= "%1s is not a valid index to delete";
	private static final String	MESSAGE_SEARCH_NO_RESULT	= "No matches for %1s in %2s";

	// Acceptable commands from user:
	enum USER_COMMAND {
		ADD, DISPLAY, DELETE, INVALID, CLEAR, SAVE, SORT, SEARCH, EXIT
	};

	private static Scanner			scanner		= new Scanner(System.in);
	private static ArrayList<String>	m_lines		= new ArrayList<String>();
	private static String			m_fileName;

	public static void main(String[] args) {
		validateArguments(args);
		processFile(args);
		showToUser(String.format(WELCOME_MESSAGE, m_fileName));
		run();
	}

	/**
	 * Method to process the file. Will set m_filename, and read from file. If
	 * file is non existant, it will create a new file.
	 *
	 * @param args
	 *		String array consisting of file name.
	 */
	private static void processFile(String[] args) {
		m_fileName = args[0];
		readFile(m_fileName);
	}

	/**
	 * method to validate arguments. The array passed in has to have at least
	 * one element to be used as filename. Otherwise, program will throw an
```

```
 * error.
 *
 * @param args
 *            The array of arguments
 */
private static void validateArguments(String[] args) {
	if (args.length == 0) {
			throw new Error("No file detected");
	}
}

/**
 * Running loop. Continuously asks user for input until user enters "exit".
 */
private static void run() {
	while (true) {
			System.out.print("command: ");
			String userCommand = scanner.nextLine();
			executeCommand(userCommand);
	}
}

/**
 * Reads the contents of file and store them in a linked list
 *
 * @param fileName
 *            name of file to read from
 */
private static void readFile(String fileName) {
	try {
			BufferedReader reader = new BufferedReader(new FileReader(fileName));
			String nextLine = reader.readLine();
			while (nextLine != null) {
					m_lines.add(nextLine);
					nextLine = reader.readLine();
			}
			reader.close();
	} catch (FileNotFoundException e) {
			newFile(fileName);

	} catch (IOException e) {
			e.printStackTrace();
	}
}

/**
 * Creates a new file with nothing inside
 *
 * @param fileName
 *            name of file to be created
 */
private static void newFile(String fileName) {
	try {
			FileWriter writer = new FileWriter(fileName);
			writer.write("");
			writer.close();
	} catch (IOException e) {
			e.printStackTrace();
	}
}

/**
 * executes the user's command. Command has to be either one of: add,
 * delete, display, clear, save or exit
 *
 * @param userCommand
 *            user's input
 */
public static void executeCommand(String userCommand) {
	validateInput(userCommand);
	USER_COMMAND commandType = determineCommandType(userCommand);
	String parameters = removeFirstWord(userCommand);
	validateParameters(parameters, commandType);

	switch (commandType) {
			case ADD :
					addText(parameters);
					break;
			case DISPLAY :
					displayContent(parameters);
					break;
			case DELETE :
					deleteText(parameters);
					break;
			case CLEAR :
					clearFile(parameters);
					break;
			case SORT :
					sortContent();
					break;
			case SEARCH :
					searchText(parameters);
					break;
			case SAVE :
					saveFile();
					break;
			case INVALID :
					showToUser(String.format(MESSAGE_INVALID_COMMAND, userCommand));
					break;
			case EXIT :
					saveFile();
					System.exit(0);
			default:
					throw new Error("Unrecognized command type");
	}
}
```

```
private static void validateInput(String input) {
	if (input.trim().equals("")) {
		throw new IllegalArgumentException(String.format(
			MESSAGE_INVALID_COMMAND, input));
	}
}

/**
 * Validates the parameters input by the user with respect to the command to
 * be executed.
 *
 * @param parameters
 *		Parameters from input line
 * @param commandType
 *		Command type of input
 */
private static void validateParameters(String parameters,
		USER_COMMAND commandType) {
	switch (commandType) {
		case ADD :
			validateAdd(parameters);
			break;
		case DELETE :
			validateDelete(parameters);
			break;
		case SEARCH :
			validateSearch(parameters);
			break;
		default:
			return;
	}

}

/**
 * Validates input parameters for add command
 *
 * @param parameters
 *		Parameters input by user
 */
private static void validateAdd(String parameters) {
	for (String line : m_lines) {
		if (parameters.equalsIgnoreCase(line)) {
			throw new IllegalArgumentException(String.format(
					MESSAGE_REPEATED_INPUT, parameters));
		}
	}
}

/**
 * validates input parameters for delete command
 *
 * @param parameters
 *		Parameters input by user
 */
private static void validateDelete(String parameters) {
	if (parameters.equals("")) {
		throw new IllegalArgumentException(String.format(
			MESSAGE_INVALID_COMMAND, parameters));
	}
	int choice = Integer.parseInt(parameters);
	if (choice > m_lines.size() || choice < 0) {
		throw new IllegalArgumentException(String.format(
			MESSAGE_WRONG_INDEX, parameters));

}

private static void validateSearch(String parameters) {
	if (parameters.equals("")) {
		throw new IllegalArgumentException(String.format(
			MESSAGE_INVALID_COMMAND, parameters));
	}
}

private static USER_COMMAND determineCommandType(String userCommand) {
	if (userCommand == null) {
		throw new Error("User command cannot be null!");
	}

	String commandType = getFirstWord(userCommand);
	if (commandType.equalsIgnoreCase("add")) {
		return USER_COMMAND.ADD;
	} else if (commandType.equalsIgnoreCase("delete")) {
		return USER_COMMAND.DELETE;
	} else if (commandType.equalsIgnoreCase("display")) {
		return USER_COMMAND.DISPLAY;
	} else if (commandType.equalsIgnoreCase("exit")) {
		return USER_COMMAND.EXIT;
	} else if (commandType.equalsIgnoreCase("clear")) {
		return USER_COMMAND.CLEAR;
	} else if (commandType.equalsIgnoreCase("save")) {
		return USER_COMMAND.SAVE;
	} else if (commandType.equalsIgnoreCase("search")) {
		return USER_COMMAND.SEARCH;
	} else if (commandType.equalsIgnoreCase("sort")) {
		return USER_COMMAND.SORT;
	} else {
		return USER_COMMAND.INVALID;
	}

}

/**
 * adds a string of text into the file
 *
 * @param userCommand
 *		should contain the command 'add' and the new line of text the
```

```
 *            user wishes to add to the file
 */
private static void addText(String parameters) {
	m_lines.add(parameters);
	showToUser(String.format(MESSAGE_FILE_ADDED, m_fileName, parameters));
}

/**
 * displays content of text file
 *
 * @param userCommand
 */
private static void displayContent(String parameters) {
	if (m_lines.size() < 1) {
			showToUser(String.format(MESSAGE_FILE_EMPTY, m_fileName));
	} else {
			for (int i = 0; i < m_lines.size(); i++) {
					showToUser((i + 1) + ". " + m_lines.get(i));
			}
	}
}

/**
 * delete text according to user's selection
 *
 * @param userCommand
 *            should contain the command delete and line number
 */
private static void deleteText(String parameters) {
	try {
			Integer choice = Integer.parseInt(parameters) - 1;
			String deleted = m_lines.get(choice);
			m_lines.remove(choice.intValue());
			if (m_lines.size() > 0) {
					showToUser(String.format(MESSAGE_FILE_DELETED, m_fileName,
							deleted));
			} else {
					showToUser(String.format(MESSAGE_FILE_EMPTY, m_fileName));
			}
	} catch (NumberFormatException nfe) {
			showToUser(String.format(MESSAGE_INVALID_COMMAND,
					parameters));
			nfe.printStackTrace();
	}
}

/**
 * clears content of text file
 *
 * @param userCommand
 */
private static void clearFile(String parameters) {
	m_lines = new ArrayList<String>();
	showToUser(String.format(MESSAGE_ALL_DELETED, m_fileName));
}

/**
 * writes the current lines into the file
 */
private static void saveFile() {
	try {
			FileWriter writer = new FileWriter(m_fileName);
			for (String lines : m_lines) {
					writer.write(lines + "\n");
			}
			writer.close();
			showToUser(String.format(MESSAGE_FILE_SAVED, m_fileName));
	} catch (IOException e) {
			showToUser(String.format(MESSAGE_FILE_IO_ERROR, m_fileName));
			e.printStackTrace();
	}
}

/**
 * Sorts the lines within the text file lexicographically. Note that numbers
 * comes before alphabets
 */
private static void sortContent() {
	Collections.sort(m_lines);
}

/**
 * Searches through the content for what the user is looking for. If not
 * found, a message will be displayed to user that there's no matches. If
 * multiple result are found, the results will be displayed in a format
 * similar to the display() method. Note that the search is case-sensitive
 * and searches for the exact sequence
 *
 * @param parameters
 *            The string user is looking for
 */
private static void searchText(String parameters) {
	int counter = 1;
	String output = "";
	boolean isfound = false;
	for (String line : m_lines) {
			if (line.contains(parameters)) {
					output = counter + ". " + line;
					showToUser(output);
					counter++;
					isfound = true;
			}
	}
	if (!isfound) {
			showToUser(String.format(MESSAGE_SEARCH_NO_RESULT, parameters,
					m_fileName));
	}
```

```
        }

        /**
         * gets the first word from a String
         *
         * @param userCommand
         *            string of text
         * @return the first word from userCommand
         */
        private static String getFirstWord(String userCommand) {
            String firstWord = userCommand.trim().split("\\s+")[0];
            return firstWord;
        }

        /**
         * Removes the first word from a String
         *
         * @param userCommand
         *            string to manipulate
         * @return userCommand without the first word
         */
        private static String removeFirstWord(String userCommand) {
            String parameters = userCommand.replace(getFirstWord(userCommand), "")
                                           .trim();
            return parameters;
        }

        private static void showToUser(String message) {
            System.out.println(message);
        }
}
```

TextBuddyTest.java

```
package cs2103;

import static org.junit.Assert.assertEquals;

import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import java.lang.reflect.Field;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class TextBuddyTest {

        private final ByteArrayOutputStream     outContent      = new ByteArrayOutputStream();
        private final ByteArrayOutputStream     errContent      = new ByteArrayOutputStream();

        private static TextBuddy                testClass       = new TextBuddy();

        @Before
        public void setUp() throws NoSuchFieldException, SecurityException,
                        IllegalArgumentException, IllegalAccessException {
            Field field = testClass.getClass().getDeclaredField("m_fileName");
            field.setAccessible(true);
            field.set(String.class, "testFile.txt");
            System.setOut(new PrintStream(outContent));
            System.setErr(new PrintStream(errContent));
            TextBuddy.executeCommand("clear");
            outContent.reset();
        }

        @After
        public void cleanUp() {
            System.setOut(null);
            System.setErr(null);
        }

        @Test
        public void testnormalAdd() {
            testOneCommand("normal add", "Added to testFile.txt: \"hello\"\n",
                           "add hello");
```

```
}

@Test
public void testAddMultipleWords() {
    testOneCommand("add with more than one word",
                   "Added to testFile.txt: \"hello world\"\n", "add hello world");
}

@Test
public void testAddSpecialChar() {
    testOneCommand("add with special characters",
                   "Added to testFile.txt: \"h!2\"\n", "add h!2");
}

@Test
public void testAddNoParam() {
    testOneCommand("add with no param", "Added to testFile.txt: \"\"\n",
                   "add ");
}

@Test
public void testClear() {
    testOneCommand("clear command",
                   "All content deleted from testFile.txt\n", "clear");
}

@Test
public void testDisplayEmpty() {
    TextBuddy.executeCommand("clear");
    outContent.reset();
    testOneCommand("display empty file", "testFile.txt is empty\n",
                   "display");
}

@Test
public void testDisplayOne() {
    TextBuddy.executeCommand("clear");
    TextBuddy.executeCommand("add hello");
    outContent.reset();
    testOneCommand("display one item", "1. hello\n", "display");
}

@Test
public void testDisplayMany() {

    TextBuddy.executeCommand("clear");
    TextBuddy.executeCommand("add hello");
    TextBuddy.executeCommand("add world");
    outContent.reset();
    testOneCommand("display multiple items", "1. hello\n2. world\n",
                   "display");
}

@Test
public void testInvalidCommand() {
    testOneCommand("Invalid command",
                   "hello is an invalid command format\n", "hello");
}

@Test
public void testSave() {
    try {
                TextBuddy.executeCommand("save");
    } catch (Exception e) {
                e.getMessage();
                assertEquals("Save command", "", errContent.toString());
    }
```

```
}

@Test
public void testDeleteOnly() {

    TextBuddy.executeCommand("clear");
    TextBuddy.executeCommand("add hello");
    outContent.reset();
    testOneCommand("delete only content in file",
                   "testFile.txt is empty\n",
                   "delete 1");
}

@Test
public void testDeleteAny() {

    TextBuddy.executeCommand("clear");
    TextBuddy.executeCommand("add hello");
    TextBuddy.executeCommand("add world");
    TextBuddy.executeCommand("add !!");
    TextBuddy.executeCommand("display");
    outContent.reset();
    testOneCommand("delete one content in file",
                   "Deleted from testFile.txt: \"world\"\n", "delete 2");
}

@Test
public void testSortFirstAlphabet() {

    TextBuddy.executeCommand("add c");
    TextBuddy.executeCommand("add a");
    TextBuddy.executeCommand("add b");
    TextBuddy.executeCommand("sort");
    outContent.reset();
    testOneCommand("Simple sorting", "1. a\n2. b\n3. c\n", "display");
}

@Test
public void testSortMultipleAlphabets() {
    TextBuddy.executeCommand("add aa a");
    TextBuddy.executeCommand("add ab a");
    TextBuddy.executeCommand("add aa b");
    TextBuddy.executeCommand("add ab b ");
    TextBuddy.executeCommand("sort");
    outContent.reset();
    testOneCommand("Detailed sorting",
                   "1. aa a\n2. aa b\n3. ab a\n4. ab b\n", "display");
}

@Test
public void testNumericSorting() {
    TextBuddy.executeCommand("add 3");
    TextBuddy.executeCommand("add 1");
    TextBuddy.executeCommand("add 2");
    TextBuddy.executeCommand("sort");
    outContent.reset();
    testOneCommand("Simple Numeric Sorting", "1. 1\n2. 2\n3. 3\n",
                   "display");
}

@Test
public void testAlphaNumericSorting() {
    TextBuddy.executeCommand("add a1");
    TextBuddy.executeCommand("add 1a");
    TextBuddy.executeCommand("sort");
    outContent.reset();
    testOneCommand("Simple AlphaNumeric Sorting", "1. 1a\n2. a1\n",
```

```
                "display");
}

@Test
public void testReverseSorting() {
    TextBuddy.executeCommand("add c");
    TextBuddy.executeCommand("add b");
    TextBuddy.executeCommand("add a");
    TextBuddy.executeCommand("sort");
    outContent.reset();
    testOneCommand("Sorting elements from reverse order",
                "1. a\n2. b\n3. c\n", "display");
}

@Test
public void testAlreadySorted() {
    TextBuddy.executeCommand("add a");
    TextBuddy.executeCommand("add b");
    TextBuddy.executeCommand("add c");
    TextBuddy.executeCommand("sort");
    outContent.reset();
    testOneCommand("Sorting elements from correct order",
                "1. a\n2. b\n3. c\n", "display");

}

@Test
public void testSimpleSearch() {
    TextBuddy.executeCommand("add The quick");
    TextBuddy.executeCommand("add brown fox");
    TextBuddy.executeCommand("add jumps over");
    outContent.reset();
    testOneCommand("Searching for one keyword", "1. brown fox\n",
                "search fox");
}

@Test
public void testAlphabetSearchOneMatch() {
    TextBuddy.executeCommand("add The quick");
    TextBuddy.executeCommand("add brown fox");
    TextBuddy.executeCommand("add jumps over");
    outContent.reset();
    testOneCommand("Searching for one alphabet", "1. brown fox\n",
                "search b");

}

@Test
public void testAlphabetSearchMultipleMatches() {
    TextBuddy.executeCommand("add The quick");
    TextBuddy.executeCommand("add brown fox");
    TextBuddy.executeCommand("add jumps over");
    outContent.reset();
    testOneCommand("Searching for one alphabet",
                "1. brown fox\n2. jumps over\n", "search o");
}

@Test
public void testSearchWithNoMatches() {
    TextBuddy.executeCommand("add The quick");
    TextBuddy.executeCommand("add brown fox");
    TextBuddy.executeCommand("add jumps over");
    outContent.reset();
    testOneCommand("Searching with no possible match",
                "No matches for a in testFile.txt\n", "search a");
}
```

```
        @Test
        public void testSearchWithTwoKeywords() {
            TextBuddy.executeCommand("add The quick brown");
            TextBuddy.executeCommand("add fox jumps over");
            TextBuddy.executeCommand("add the lazy dog");
            outContent.reset();
            testOneCommand("Searching with two keywords",
                           "1. The quick brown\n", "search The quick");
        }

        public void testOneCommand(String description, String expected,
                        String command) {
            TextBuddy.executeCommand(command);
            assertEquals(description, expected, outContent.toString());
        }
}
```

CE1 code

```
package cs2103;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.LinkedList;
import java.util.Scanner;

/**
 * This class is used to manipulate a text file. It supports 6 basic commands:
 * add, delete, display, clear, save and exit
 *
 * @author Victor Hazali
 */
public class TextBuddy {

          private static final String   WELCOME_MESSAGE                       = "Welcome to TextBuddy. %1$s is ready for use";
          private static final String   MESSAGE_FILE_DELETED          = "Deleted from %1$s: \"%2$s\"";
          private static final String   MESSAGE_ALL_DELETED           = "All content deleted from %1$s";
          private static final String   MESSAGE_FILE_ADDED            = "Added to %1$s: \"%2$s\"";
          private static final String   MESSAGE_FILE_EMPTY            = "%1$s is empty";
          private static final String   MESSAGE_INVALID_COMMAND       = "%1$s is an invalid command format";
          private static final String   MESSAGE_FILE_IO_ERROR         = "Failed to open %1$s";
          private static final String   MESSAGE_FILE_SAVED            = "%1$s successfully saved";

          // Acceptable commands from user:
          enum USER_COMMAND {
                    ADD, DISPLAY, DELETE, INVALID, CLEAR, SAVE, EXIT
          };

          private static Scanner                                      scanner   = new Scanner(System.in);
          private static LinkedList<String>       m_lines   = new LinkedList<String>();
          private static String                                       m_fileName;

          public static void main(String[] args) {
                    m_fileName = args[0];
                    readFile(m_fileName);
                    System.out.println(String.format(WELCOME_MESSAGE, m_fileName));
                    while (true) {
                              System.out.print("command: ");
                              String userCommand = scanner.nextLine();
                              executeCommand(userCommand);
                    }
          }

          /**
           * Reads the contents of file and store them in a linked list
           *
           * @param fileName
           *            name of file to read from
           */
          private static void readFile(String fileName) {
                    try {
                              BufferedReader reader = new BufferedReader(new FileReader(fileName));
                              String nextLine = reader.readLine();
                              while (nextLine != null) {
                                        m_lines.add(nextLine);
                                        nextLine = reader.readLine();
                              }
```

```
                reader.close();
        } catch (FileNotFoundException e) {
                newFile(fileName);

        } catch (IOException e) {
                e.printStackTrace();
        }
}

/**
 * Creates a new file with nothing inside
 *
 * @param fileName
 *            name of file to be created
 */
private static void newFile(String fileName) {
        try {
                FileWriter writer = new FileWriter(fileName);
                writer.write("");
                writer.close();
        } catch (IOException e) {
                e.printStackTrace();
        }
}

/**
 * executes the user's command. Command has to be either one of: add,
 * delete, display, clear, save or exit
 *
 * @param userCommand
 *            user's input
 */
private static void executeCommand(String userCommand) {
        if (userCommand.trim().equals("")) {
                System.out.println(String.format(MESSAGE_INVALID_COMMAND,
                                userCommand));
                return;
        }

        USER_COMMAND commandType = determineCommandType(userCommand);

        switch (commandType) {
                case ADD :
                        addText(userCommand);
                        break;
                case DISPLAY :
                        displayContent(userCommand);
                        break;
                case DELETE :
                        deleteText(userCommand);
                        break;
                case CLEAR :
                        clearFile(userCommand);
                        break;
                case SAVE :
                        saveFile();
                        break;
                case INVALID :
                        System.out.println(String.format(MESSAGE_INVALID_COMMAND,
                                        userCommand));
                        break;
                case EXIT :
                        saveFile();
                        System.exit(0);
                default:
                        throw new Error("Unrecognized command type");
        }
}

private static USER_COMMAND determineCommandType(String userCommand) {
        if (userCommand == null) {
                throw new Error("User command cannot be null!");
        }

        String commandType = getFirstWord(userCommand);
        if (commandType.equalsIgnoreCase("add")) {
                return USER_COMMAND.ADD;
        } else if (commandType.equalsIgnoreCase("delete")) {
                return USER_COMMAND.DELETE;
        } else if (commandType.equalsIgnoreCase("display")) {
                return USER_COMMAND.DISPLAY;
        } else if (commandType.equalsIgnoreCase("exit")) {
                return USER_COMMAND.EXIT;
        } else if (commandType.equalsIgnoreCase("clear")) {
                return USER_COMMAND.CLEAR;
        } else if (commandType.equalsIgnoreCase("save")) {
                return USER_COMMAND.SAVE;
        } else {
                return USER_COMMAND.INVALID;
        }
```

```
}

/**
 * adds a string of text into the file
 *
 * @param userCommand
 *            should contain the command 'add' and the new line of text the
 *            user wishes to add to the file
 */
private static void addText(String userCommand) {
          String text = removeFirstWord(userCommand);
          m_lines.add(text);
          System.out.println(String.format(MESSAGE_FILE_ADDED, m_fileName, text));
}

/**
 * displays content of text file
 *
 * @param userCommand
 */
private static void displayContent(String userCommand) {
          if (m_lines.size() < 1) {
                    System.out.println(String.format(MESSAGE_FILE_EMPTY, m_fileName));
          } else {
                    for (int i = 0; i < m_lines.size(); i++) {
                              System.out.println((i + 1) + ". " + m_lines.get(i));
                    }
          }
}

/**
 * delete text according to user's selection
 *
 * @param userCommand
 *            should contain the command delete and line number
 */
private static void deleteText(String userCommand) {
          String text = removeFirstWord(userCommand);
          try {
                    Integer choice = Integer.parseInt(text) - 1;
                    String deleted = m_lines.get(choice);
                    m_lines.remove(choice.intValue());
                    System.out.println(String.format(MESSAGE_FILE_DELETED, m_fileName,
                                        deleted));
          } catch (NumberFormatException nfe) {
                    System.out.println(String.format(MESSAGE_INVALID_COMMAND,
                                        userCommand));
          }
}

/**
 * clears content of text file
 *
 * @param userCommand
 */
private static void clearFile(String userCommand) {
          m_lines = new LinkedList<String>();
          System.out.println(String.format(MESSAGE_ALL_DELETED, m_fileName));
}

/**
 * writes the current lines into the file
 */
private static void saveFile() {
          try {
                    FileWriter writer = new FileWriter(m_fileName);
                    for (String lines : m_lines) {
                              writer.write(lines + "\n");
                    }
                    writer.close();
                    System.out.println(String.format(MESSAGE_FILE_SAVED, m_fileName));
          } catch (IOException e) {
                    System.out
                                        .println(String.format(MESSAGE_FILE_IO_ERROR, m_fileName));
          }
}

/**
 * gets the first word from a String
 *
 * @param userCommand
 *            string of text
 * @return the first word from userCommand
 */
private static String getFirstWord(String userCommand) {
          String firstWord = userCommand.trim().split("\\s+")[0];
          return firstWord;
}

/**
```

```
 * Removes the first word from a String
 *
 * @param userCommand
 *            string to manipulate
 * @return userCommand without the first word
 */
private static String removeFirstWord(String userCommand) {
          String parameters = userCommand.replace(getFirstWord(userCommand), "")
                              .trim();
          return parameters;
}
```