```java
/*

 HACK ASSEMBLER - translates assembly (foo.asm) into Hack machine language (foo.hack).

*/

package assembler;

import java.io.*;
import java.util.*;

public class HackAssembler {

    // In order to read the .asm file and write to the .hack file
    private BufferedReader r;
    private PrintWriter w;

    // The three other components of the assembler
    private final SymbolTable symbolTable = new SymbolTable(); // Contains the mapping of
symbolic references (predefined symbols, labels, static variables). Used for A-
instructions.
    private final AssemblyParser parse = new AssemblyParser(); // Decomposes each
assembly instruction and retrieves its various parts.
    private final BinaryTable translate = new BinaryTable();   // Contains the mapping of
each part of a C-instruction with the corresponding binary code.

    // Loaded file internal representation.
    private static String assemblyFile;                    // Contains the full path of
the .asm file
    private final Vector<String> program = new Vector<>(); // Contains the entire .asm
file inside the program, in order to easily double-pass it.
    private String line;                                   // Current line being
processed.


    public HackAssembler(String file) {
        assemblyFile = file;

        loadFile();   // Loads the .asm file into "program" Vector<String>
        firstPass();  // First pass, loads labels.
        secondPass(); // Second pass, loads static variables and does the actual
translation.
        exit();       // Closes r and w.
    }


    private void loadFile() {
        initializeIO(); // Creates reader and writer to .asm and .hack files respectively.
        try {
            // Cycle the .asm file line by line and load it into "program" Vector<String>.
            while(true) {
                line = r.readLine();
                if (line == null) break;               // File ended, exit the while.
                clean();                               // Removes spaces and comments
from each line.
                if(!line.isEmpty()) program.add(line); // Removes empty lines
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void initializeIO() {
        try {
            r = new BufferedReader(new FileReader(new File(assemblyFile)));
            w = new PrintWriter(new FileWriter(new File(assemblyFile.replaceAll(".asm",
```

```
".hack"))));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void clean() {
        line = line.replaceAll(" ", ""); // Removes spaces.
        line = removeComments(line);                    // Removes comments.
    }

    private String removeComments(String line) {
        int index = line.indexOf("//");
        if(index != -1) line = line.substring(0, index); // In case there is a comment,
it trims it out.
        return line;
    }

    private void firstPass() {
        for(int i=0; i<program.size(); i++) {
            line = program.elementAt(i);
            int index1 = line.indexOf("(");
            int index2 = line.indexOf(")");
            if(index1 != -1 && index2 != -1) {                  // If I find a label
(there are parentheses)
                String label = line.substring(index1 + 1, index2); // Get the name of the
label trimming out the parentheses.
                symbolTable.add(label, i);                      // The label gets
mapped to its own line number, because after the removal of the label line, it will be
the line of the first instruction after the label.
                program.removeElementAt(i);                     // Removes label line.
                i--;                                            // I have to re-
analyze the same line because it now contains the next instruction after the label.
            }
        }
    }

    private void secondPass() {
        for(int i=0; i<program.size(); i++) {
            line = program.elementAt(i);
            if (line.indexOf("@") == 0) handleAInstruction(); // Distinguishes between A-
instruction and C-instruction.
            else handleCInstruction();
        }
    }

    private void handleAInstruction() { // @value -> 0xxxxxxxxxxxxxxx where '0'
identifies the A-instruction and xxxxxxxxxxxxxxx the relative value (address)
        int value;
        String valueString = parse.AInstructionInt(line);
        if (isNumber(valueString)) value = Integer.parseInt(valueString); // Just
extracts the value number if "value" is not a variable.
        else value = symbolTable.retrieveValue(valueString);                        //
Returns the cell memory number assigned to the variable, whether it is an old or novel
variable. If it is just a

        String binaryValue = Integer.toBinaryString(value); // Takes for granted that no
addresses greater than 15-bit are even written and translates to binary.
        while (binaryValue.length()<16) binaryValue = "0" + binaryValue; // The final
result is just the binary version of 'value' with enough zeros before it to make it 16-
bit.
        w.println(binaryValue);
    }

    private void handleCInstruction() { // dest = comp ; jump   ->   111accccccdddjjj is
the binary structure of a C-instruction
```

```
        String dest = parse.dest(line);
        String comp = parse.comp(line);
        String jump = parse.jump(line);

        String accccc = translate.comp(comp);
        String ddd = translate.dest(dest);
        String jjj = translate.jump(jump);

        w.println("111" + accccc + ddd + jjj); // Composes the binary instruction
    }

    public boolean isNumber(String valueString) {
        try {
            Integer.parseInt(valueString);
        } catch (NumberFormatException e) { // If it isn't a number, it is a variable.
            return false;
        }
        return true;
    }

    public void exit() {
        try{
            w.close();
            r.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```