```java
import java.io.File;
// import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;

public class HackAssembler
{
  public static void main(String[] args)
  {
    // Initialize
    // Opens the input file (Prog.asm) and gets ready to process it
    String filename;
    if (args.length == 0)
    {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter filename (+ extension): ");
      filename = sc.nextLine();
      sc.close();
    }
    else
    {
      filename = args[0];
    }

    File ogFile = new File(filename);
    Scanner scFile, hFile;
    try
    {
      scFile = new Scanner(ogFile);
      System.out.println("Translating " + filename + " into hack code.\nContents of file:\n");

      // create output file
      String outFilename = filename.substring(0, filename.lastIndexOf(".")) + ".hack";
      File hackFile = new File(outFilename);

      SymbolTable sTable = new SymbolTable();
      // do first pass
      firstPass(ogFile, sTable);

      // do second pass
      secondPass(ogFile, hackFile, sTable);

      hFile = new Scanner(new File(outFilename));
      while (hFile.hasNextLine())
        System.out.println(hFile.nextLine());

      // close files
      scFile.close();

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

  }
```

```
58
59    /**
60     * Reads the program lines, one by one focusing only on (label) declarations.
61     * Adds the found labels to the symbol table
62     *
63     * @throws IOException
64     */
65    public static void firstPass(File inFile, SymbolTable sTable) throws IOException
66    {
67      Parser p = new Parser(inFile);
68      int addr = 0;
69
70      while (p.hasMoreCommands())
71      {
72        p.advance();
73
74        Parser.CommandType instructionType = p.instructionType();
75
76        if (instructionType == null)
77          throw new IllegalStateException("Syntax error at instruction " + (addr +
   1));
78
79        switch (instructionType)
80        {
81          case A_COMMAND:
82          case C_COMMAND:
83            addr++;
84            break;
85          case L_COMMAND:
86            String symbol = p.symbol();
87
88            if (Character.isDigit(symbol.charAt(0)))
89              throw new IllegalStateException("Symbol syntax error at instruction " +
   (addr + 1));
90
91            // add label to symbol table
92            sTable.addEntry(symbol, addr);
93        }
94      }
95    }
96
97    /**
98     * While there are more lines to process: Gets the next instruction, and parses
99     * it If the instruction is @symbol If symbol is not in the symbol table, adds
100    * it Translates the symbol into its binary value If the instruction is dest
101    * =comp ; jump Translates each of the three fields into its binary value
102    * Assembles the binary values into a string of sixteen 0’s and 1’s Writes the
103    * string to the output file.
104    *
105    * @throws IOException
106    */
107   public static void secondPass(File inFile, File outFile, SymbolTable sTable)
   throws IOException
108   {
109     Parser p = new Parser(inFile);
110     int currentAddr = 16;
111
112     FileWriter fileWrite = null;
113     try
114     {
```

```
115       fileWrite = new FileWriter(outFile);
116     }
117     catch (IOException e)
118     {
119       e.printStackTrace();
120     }
121 
122     // fileWrite.write("TEST FILE WRITER\nTHIS IS A NEWLINE");
123 
124     String resultOfLine;
125 
126     while (p.hasMoreCommands())
127     {
128       p.advance();
129 
130       switch (p.instructionType())
131       {
132         case A_COMMAND:
133           String symbol = p.symbol();
134           boolean isDecimal = Character.isDigit(symbol.charAt(0));
135 
136           // add symbol to table if it is not a symbol
137           if (!isDecimal && !sTable.contains(symbol))
138             sTable.addEntry(symbol, currentAddr++);
139 
140           // get the integer value of the symbol
141           int value = isDecimal ? Integer.parseInt(symbol) :
sTable.getAddress(symbol);
142 
143           resultOfLine = "0" + String.format("%15s",
Integer.toBinaryString(value)).replaceAll(" ", "0") + "\n";
144 
145           try
146           {
147             fileWrite.append(resultOfLine);
148           }
149           catch (IOException e)
150           {
151             throw new IllegalStateException(e.getMessage());
152           }
153 
154           break;
155         case C_COMMAND:
156           // get binary of comp, dest, and jump
157           String comp = Code.comp(p.comp());
158           String dest = Code.dest(p.dest());
159           String jump = Code.jump(p.jump());
160 
161           resultOfLine = "111" + comp + dest + jump + "\n";
162 
163           try
164           {
165             fileWrite.write(resultOfLine);
166           }
167           catch (IOException e)
168           {
169             throw new IllegalStateException(e.getMessage());
170           }
171 
172           break;
```

```
173         default:
174       }
175     }
176 
177     try
178     {
179       fileWrite.close();
180     }
181     catch (IOException e)
182     {
183       throw new IllegalStateException(e.getMessage());
184     }
185   }
186 }
```