Step 2: Set Up a New Java Project

1. Open Eclipse and go to the **File** menu.
2. Select **New** > **Java Project**.
3. Name the project (e.g., `AssemblerPass1`), and click **Finish**.

Step 3: Create the Java Classes

1. Right-click on the `src` folder in your project in the Project Explorer.
2. Select **New** > **Class**.
3. Create the following classes one by one:
   - `APass1`
   - `Obj`
   - `Pooltable`

Step 4: Paste the Code

1. Paste the code for `APass1`, `Obj`, and `Pooltable` into the corresponding class files.
   - For `APass1`, paste the **main assembler logic**.
   - For `Obj`, paste the **Obj class** (used to store symbol or literal data).
   - For `Pooltable`, paste the **Pooltable class** (used to store pool information).

Step 5: Configure the Input File

1. Ensure that the `Sample.txt` file (which contains the assembly code) exists at the path `C:\\Eclips\\TCOB06\\Pass1\\Sample.txt`. You can either:
   - Create this directory structure and the text file.
   - Change the file path in the Java code to match the location where your `Sample.txt` file is stored.

Step 6: Run the Program

1. Right-click on the `APass1.java` file in the **Project Explorer** and select **Run As** > **Java Application**.

Step 7: View Output

1. The output will be generated in the **Console** window.
2. Additionally, the output will be written to a file named `Output.txt` as specified in the code.
3. You can view the `Output.txt` file from the file system or read it from the **Console** output.

Overview of the Program

This program is the **First Pass** of a two-pass assembler. It processes an assembly language source code line by line, building tables to track symbols, literals, and instructions. These tables will be used in the second pass to generate the machine code.

Key Concepts

1. **Symbol Table**: This table holds symbols (such as variables or labels) and their corresponding memory addresses. It helps the assembler know where to place each symbol in memory.
2. **Literal Table**: This table keeps track of constants (like numeric literals) that appear in the source code. Constants are assigned memory addresses for later replacement.
3. **Pool Table**: This table is used to manage "pools" of literals that need to be processed in chunks. It helps efficiently handle literals by grouping them together.
4. **Opcode Table (Optab)**: This table maps the mnemonics (like `add`, `sub`, `mult`) to their respective opcode numbers. This is essential for translating assembly instructions into machine code.
5. **Location Counter (Loc)**: This counter tracks the current memory location where the next instruction or data will be placed. It helps the assembler allocate memory addresses for instructions and data sequentially.

Working of the Program

1. File Handling:

- The program reads the assembly source code from a file (`Sample.txt`) line by line using a `BufferedReader`.
- The output (processed results) is written to another file (`Output.txt`) using a `BufferedWriter`.

2. Parsing Assembly Instructions:

The program processes each line of the assembly code and breaks it down into tokens (components of the instruction). Each token is then categorized into one of several types such as:

- **Instruction** (like `add`, `mover`, `sub`)
- **Data Definitions** (like `ds`, `dc`)
- **Symbol Definitions** (like `up:`)
- **Literal Values** (like `'5'`, `'7'`)
- **Directives** (like `start`, `end`, `origin`, `ltorg`, `equ`)

3. Handling Directives:

The program recognizes different assembler directives like `start`, `end`, `origin`, etc. These directives instruct the assembler on how to process the code:

- **start**: Marks the beginning of the program and sets the starting address for the location counter.
- **end**: Marks the end of the program. It also processes any remaining literals and symbols.

- **origin**: Used to set the location of a label (symbol).
- **ltorg**: Directs the assembler to process all pending literals and assign them memory addresses.
- **equ**: Defines a constant symbol, assigning it the address of another symbol.

4. Processing Symbols:

The program processes labels (like `up:`) by searching the symbol table. If the symbol isn't found, it is added to the symbol table with its memory address.

- **Symbol Table** keeps track of labels and their corresponding memory locations.

5. Processing Literals:

Literals are constant values in the program (like `'5'`, `'7'`). When the assembler encounters a literal, it adds it to the literal table and assigns it a memory location.

- **Literal Table** keeps track of the literals encountered during the pass and their addresses.

6. Memory Allocation (Location Counter):

- **Location Counter** is incremented as the program reads each line. Each time an instruction, literal, or data definition is processed, the counter advances to allocate the next available memory location.

7. Generating Output:

The output includes:

- **Symbol Table**: Lists all symbols with their corresponding memory addresses.
- **Pool Table**: Groups literals into pools and tracks how many literals belong to each pool.
- **Literal Table**: Lists all literals with their assigned memory addresses.
- **Opcode Table**: Displays the mnemonic opcodes with their corresponding values.

8. Error Handling:

If there is an issue reading the file, the program prints an error message and terminates. Errors are caught using try-catch blocks.

9. Searching for Tokens:

To process instructions, symbols, and literals, the program uses helper functions to search through predefined lists of valid instructions (IS), data types (DL), and registers (REG). It checks if a token matches any of the known valid items and processes it accordingly.

10. Symbol, Literal, and Pool Table Updates:

- As the program processes each token, it updates the symbol table for labels, the literal table for constants, and the pool table for groupings of literals.

- For example, when the program encounters a label like `up:`, it will check if this label already exists in the symbol table; if not, it will add it with the current location counter value.
- For literals (such as `='5'`), the program stores the literal in the literal table and assigns it a memory location.

Example Workflow

Let's say the source code contains the following lines:

```
plaintext
Copy code
start 100
mover ax 05
mover bx 10
up: add ax bx
movem a ='5'
```

- **start 100**: The location counter (`loc`) is set to 100, marking the start of the program.
- **mover ax 05**: The instruction `mover` is identified and mapped to its opcode (using the opcode table). The register `ax` is identified, and the literal `05` is placed in the literal table.
- **up: add ax bx**: The label `up:` is added to the symbol table with the current location counter value (e.g., 101). The instruction `add ax bx` is processed using the opcode table, with the registers `ax` and `bx` identified.
- **movem a ='5'**: The literal `5` is added to the literal table, and memory is allocated.

This process continues until the end of the code, building the necessary tables (symbol, literal, pool) and producing the assembler output.

Output

The output would show the processed tables like this:

1. **Symbol Table**:

   ```
   css
   Copy code
   up    101
   a     102
   b     103
   c     104
   ```

2. **Literal Table**:

   ```
   arduino
   Copy code
   1    '5'    105
   2    '10'   106
   ```

3. **Pool Table**:

   ```
   Copy code
   1    2
   ```

4. **Opcode Table**:

    ```
    csharp
    Copy code
    mover    1
    add      2
    ```

This output helps the second pass of the assembler to generate the actual machine code by referring to the addresses and opcodes stored in these tables.

Conclusion

The first pass of an assembler doesn't generate machine code. Instead, it builds the necessary tables (symbol table, literal table, pool table, and opcode table) and performs memory allocation, all of which will be referenced in the second pass to generate the actual object code. The program processes the input line by line, categorizes tokens, and manages memory allocation, while also handling errors and generating informative output for debugging and further processing.