Create a New Project in Eclipse:

- Open Eclipse.
- Go to **File** > **New** > **Java Project**.
- Enter a name for your project (e.g., `AssemblerPass2`).
- Click **Finish** to create the project.

3. Create the Java Classes:

- Right-click on the `src` folder within the newly created project in the Project Explorer.
- Select **New** > **Class**.
- Name the class `Pass2` and make sure the checkbox for `public static void main(String[] args)` is selected. Click **Finish**.
- Now, you should see the `Pass2.java` class inside the `src` folder.
- Repeat this process to create the `Obj.java` class.

4. Add Code to the Classes:

- Copy the `Pass2.java` code into the `Pass2.java` class.
- Copy the `Obj.java` code into the `Obj.java` class.

5. Provide the `Output.txt` File:

- The program requires an `Output.txt` file located at `C:\\Eclips\\TCOB06\\A_Pass2\\Output.txt`.
- Make sure you have a file named `Output.txt` with the contents provided in your program's example.
- If you're just testing, you can create a similar file or adjust the code to match your file path.

6. Run the Program:

- Click the **Run** button (green play button) in Eclipse, or right-click on the `Pass2.java` file and select **Run As** > **Java Application**.

7. View the Output:

- The output should be displayed in the **Console** tab in Eclipse.
- If any issues occur, check the console for errors and debug accordingly.

Detailed Explanation of the Code

The provided `Pass2.java` code is a simple simulation of an assembler's second pass. The second pass of an assembler deals with translating symbolic addresses into actual memory addresses, generating machine code from the intermediate representation of source code. Let's break down the functionality and the flow of the code:

1. Class Definition (`Pass2.java`):

- The class `Pass2` contains the main program logic for the second pass of an assembler.

- It works with two important data structures:
  - **Symbol Table:** Holds the names of symbols and their corresponding memory addresses.
  - **Literal Table:** Holds the literals (like constants or numbers) and their memory addresses.

2. Data Structures:

- `Obj[] symb_table = new Obj[10];` and `Obj[] literal_table = new Obj[10];`:
  - Arrays to store symbol and literal tables.
  - Each `Obj` object contains the `name` (symbol or literal) and `addr` (address).
- `symb_found = 0;`: A counter used to track whether any duplicate symbols are found.

3. Reading Input:

- **Input for Symbol Table:**
  - First, the program asks the user to enter the number of symbols.
  - Then for each symbol, it collects the symbol's name and address.
- **Input for Literal Table:**
  - Similarly, it collects the number of literals and their names and addresses.
- These inputs are stored in the `symb_table` and `literal_table` arrays, respectively.

4. Displaying Symbol and Literal Tables:

- After collecting the input, the program prints the content of the symbol table and literal table to the console.
  - Symbol Table shows the symbol names and their addresses.
  - Literal Table shows the index, literal name, and address.

5. Reading and Processing the Output File (`Output.txt`):

- The program then reads an output file (`Output.txt`) line by line.
- The lines are processed as follows:
  - **Tokens** are separated by spaces.
  - The tokens are evaluated to check whether they correspond to symbols, literals, or certain mnemonics.
- **Address Processing:**
  - If the token is a number, it is treated as a location counter (or memory address).
  - If the token is a symbol (e.g., `S`), the program attempts to match it with the symbol table and print the corresponding address.
  - If the token is a literal (e.g., `L`), the program does the same with the literal table.
  - If no valid address is found, it prints a symbol error (`---`).

6. Symbol Duplication Check:

- After processing the file, the program checks for duplicate symbols in the symbol table.
- If any duplicate is found, it prints a warning indicating the duplicate symbol.

7. Error Handling:

- The program checks for various types of errors:
    - **Symbol Not Defined:** If a symbol doesn't exist in the symbol table, it prints an error.
    - **Invalid Mnemonic:** If an invalid mnemonic is found, it prints a message indicating the invalid mnemonic.

8. Final Output:

- Finally, the program generates an output that shows the final processed code, including symbol and literal replacements.

Key Concepts and How They Work:

- **Symbol Table:** It is a mapping between a symbolic name (like a variable or label in assembly code) and a memory address. During the second pass, the assembler replaces these symbolic names with actual addresses.
- **Literal Table:** Similar to the symbol table, but it stores constants and literals used in the code.
- **Mnemonic Parsing:** The program recognizes mnemonics and processes them accordingly. For example, `S` for symbol, `L` for literal, `AD` for address, and `C` for constant.
- **Duplicate Symbol Check:** Ensures that no symbol is defined multiple times, which can cause errors in the assembly process.