Step 2: Set Up a New Java Project in Eclipse

1. Open Eclipse and create a new project:
   - Go to `File` > `New` > `Java Project`.
   - Name the project (e.g., `MacroPass2`).
2. Click `Finish` to create the project.

Step 3: Add the Java Files

1. Create a new package within the `src` folder:
   - Right-click on `src` > `New` > `Package` > Name it `pkg1`.
2. Inside the `pkg1` package, create the following Java classes:
   - `macropass2.java` (main program)
   - `arglist.java` (for argument handling)
   - `mdt.java` (for macro definition table)
   - `mnt.java` (for macro name table)
3. Copy and paste the content of the respective Java code files into the newly created classes.

Step 4: Add the Text Files

1. Inside the project folder (`MacroPass2`), create a folder named `src` and add the following text files:
   - `MNT.txt`
   - `MDT.txt`
   - `argmnt.txt`
   - `input.txt`
   - Ensure the paths in the code match the locations where you place these files.

Step 5: Run the Program

1. Right-click on the `macropass2.java` file and select `Run As` > `Java Application`.
2. The program should execute, and you'll see the output in the console and output.txt file.

Detailed Explanation of the Code

The given code is an implementation of **Macro Pass 2** of a macro processor, which processes assembly-like macro instructions and expands them into the actual code. Below is a breakdown of how the different components of the code work.

Key Classes and Their Roles:

1. **macropass2.java (Main Program)**
   - This class reads the macro-related data, expands macros into actual instructions, and writes the results to the output file.
2. **arglist.java**
   - This class is used to store arguments. Each `arglist` object holds an argument's name and value (if provided).
3. **mdt.java**
   - Represents a **Macro Definition Table (MDT)** entry. Each `mdt` object stores a macro statement.
4. **mnt.java**

- Represents a **Macro Name Table (MNT)** entry. Each `mnt` object stores a macro name, its address in the MDT, and the number of arguments it takes.

Explanation of the Main Logic in `macropass2.java`:

1. **Reading Input Files**:
   - The program reads from three main files: `MNT.txt`, `MDT.txt`, and `argmnt.txt`. These files contain macro-related data, such as the macro names and their associated definitions.
   - **MNT.txt** is loaded into the `MNT` array, where each entry contains the macro name, its address, and the number of arguments it takes.
   - **argmnt.txt** is read into an `arglist` array, which stores the formal parameters of the macros.
   - **MDT.txt** is read into the `MDT` array, which contains the actual macro instructions.
2. **Macro Expansion Process**:
   - The program processes the `input.txt` file, which contains code with macro calls.
   - It identifies when a macro is invoked and then expands it using the data from the **MDT** and **MNT**.
3. **Macro Invocation and Argument Replacement**:
   - When a macro is encountered (indicated by the `macro_start` flag), the program looks up the corresponding macro definition in the **MDT**.
   - The actual arguments from `input.txt` (such as `N1`, `N2`) are substituted for the formal parameters defined in `argmnt.txt`.
4. **Generating Output**:
   - The macro is expanded by replacing formal parameters (e.g., `&X`, `&Y`) with the actual arguments. The program writes the expanded code into `output.txt`.
5. **Flags**:
   - The `macro_start`, `macro_end`, and `macro_call` flags are used to track whether a macro is currently being processed and to store information about which macro is being expanded.

Key Operations:

- **Reading Macro Name Table (MNT)**:
  - The `MNT.txt` file is read to build the macro name table (`MNT`), where each entry contains the macro name, its address, and the number of arguments it requires.
- **Reading Formal Parameters**:
  - The `argmnt.txt` file is read to build the list of formal parameters for the macros.
- **Reading Macro Definition Table (MDT)**:
  - The `MDT.txt` file is read to build the macro definition table (`MDT`), where each entry contains a macro instruction.
- **Processing Input (Macro Expansion)**:
  - The program reads `input.txt` and processes each line:
    - If a macro is encountered (using the keyword `MACRO`), the program identifies it as a macro definition.
    - The arguments are passed as needed when the macro is called.
    - The actual code for the macro is written to the output file by replacing placeholders with actual arguments.
- **Output Generation**:

- Once the macro has been expanded, the program writes the final code (with macros replaced) into `output.txt`.

Example Walkthrough:

For example, in the `input.txt` file:

```
sql
Copy code
MACRO
INCR &X, &Y, &REG1 = AREG
MOVER &REG1, &X
ADD &REG1, &Y
MOVEM &REG1, &Y
MEND
START 100
READ N1
READ N2
INCR N1, N2
STOP
N1 DS 1
N2 DS 2
END
```

- The `MACRO` statement triggers the beginning of a macro definition.
- The macro `INCR` is defined with formal parameters `&X`, `&Y`, and `&REG1`.
- Later in the `input.txt` file, the `INCR` macro is called with actual arguments `N1`, `N2`.
- The macro gets expanded in the output with the actual parameters replacing the formal ones.

After processing, the **expanded code** would be written to `output.txt`, replacing the macro invocation `INCR N1, N2` with the actual macro instructions expanded with `N1` and `N2`.

Summary:

The code simulates **Macro Pass 2** of a macro processor, which is part of the compilation process that handles macro instructions. It performs the following tasks:

1. Reads macro definitions and formal arguments.
2. Expands macros by replacing formal arguments with actual arguments.
3. Outputs the expanded code to a file.

This is a typical example of a **macro processing system** used in compilers and assemblers.