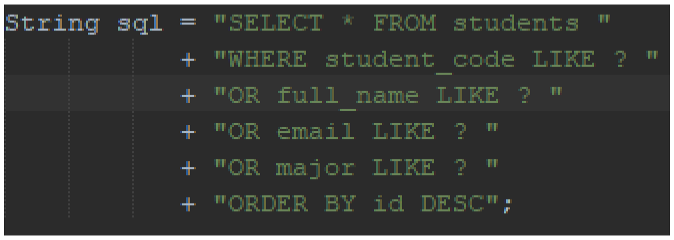
Ex5.1:

searchStudents(String keyword) method:

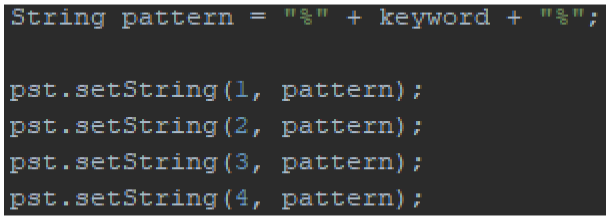


```
String sql = "SELECT * FROM students "
           + "WHERE student_code LIKE ? "
           + "OR full_name LIKE ? "
           + "OR email LIKE ? "
           + "OR major LIKE ? "
           + "ORDER BY id DESC";
```

This SQL query searches 4 columns:

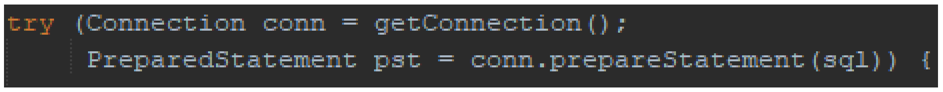
- student_code
- full_name
- email
- major

it uses “like” so partial matches are allowed. “ORDER BY id DESC” means newest students will appear first.



```
String pattern = "%" + keyword + "%";

pst.setString(1, pattern);
pst.setString(2, pattern);
pst.setString(3, pattern);
pst.setString(4, pattern);
```

This warps the keyword with % to allow partial matching and sets the same pattern for all 4 search fields.

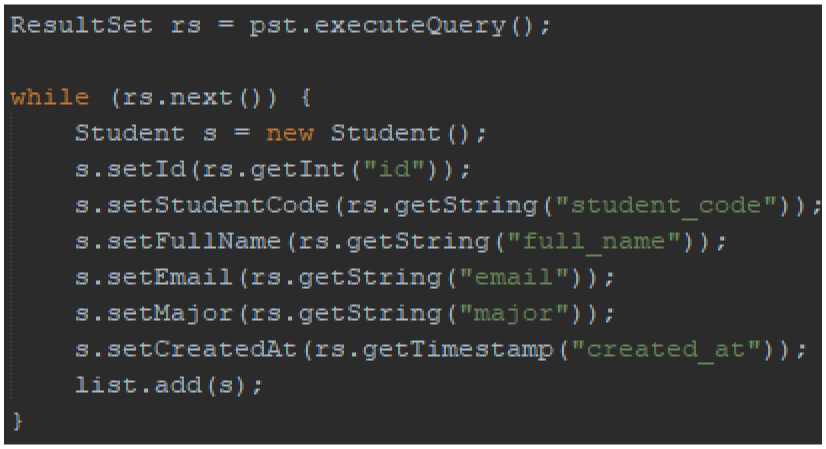


```
try (Connection conn = getConnection();
     PreparedStatement pst = conn.prepareStatement(sql)) {
```

Automatically closes:

- Connection
- PreparedStatement
- ResultSet

Prevents memory leaks



```java
ResultSet rs = pst.executeQuery();

while (rs.next()) {
    Student s = new Student();
    s.setId(rs.getInt("id"));
    s.setStudentCode(rs.getString("student_code"));
    s.setFullName(rs.getString("full_name"));
    s.setEmail(rs.getString("email"));
    s.setMajor(rs.getString("major"));
    s.setCreatedAt(rs.getTimestamp("created_at"));
    list.add(s);
}
```

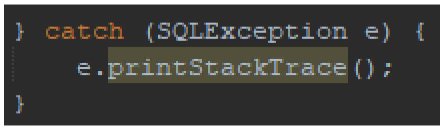
executeQuery() sends the SQL to the database.

The “while” loop iterates through each matching row.

A new “Student” object is created for every row.

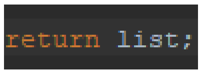
All column values are extracted and set using getters.

The student is added to the result “List<Student>”



```java
} catch (SQLException e) {
    e.printStackTrace();
}
```

If the query fails, it prints the stack trace.

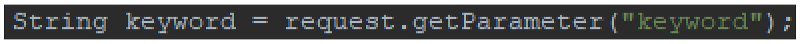


```java
return list;
```

This gives the controller a list of all matching students.

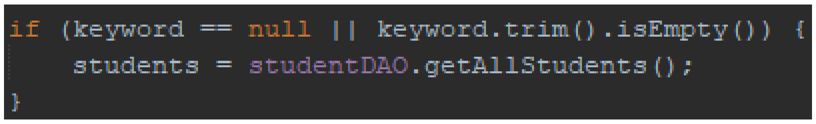
Ex5.2:

searchStudents(request, response) method:



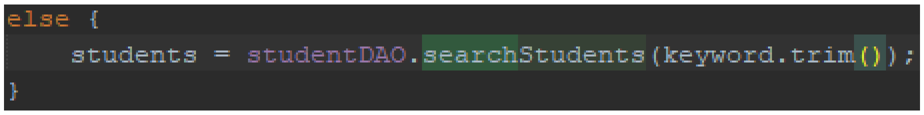
```java
String keyword = request.getParameter("keyword");
```

Gets whatever the user typed into the search bar.



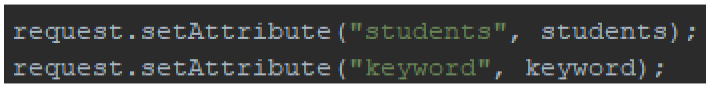
```java
if (keyword == null || keyword.trim().isEmpty()) {
    students = studentDAO.getAllStudents();
}
```

If nothing is typed, all students will be listed.



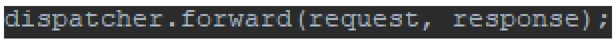
```
else {
    students = studentDAO.searchStudents(keyword.trim());
}
```

Calls the DAO to find students whose “code”, “name”, “email”, or “major” contain the keyword.



```
request.setAttribute("students", students);
request.setAttribute("keyword", keyword);
```

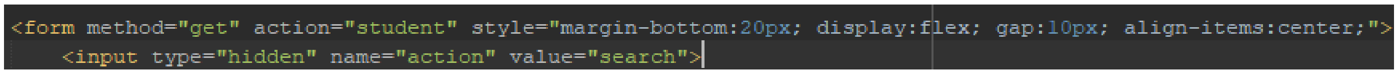
Stores data for JSP so that the JSP can show results and keep the search text in the input field.



```
dispatcher.forward(request, response);
```

Renders the same student list page, but with filtered result.

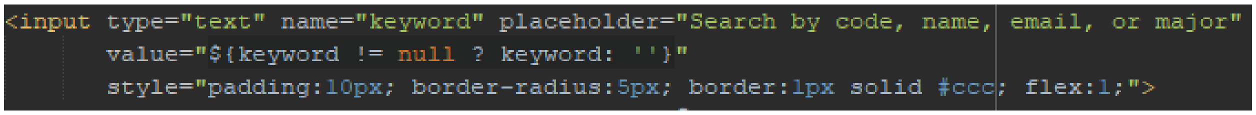
Ex5.3:



```
<form method="get" action="student" style="margin-bottom:20px; display:flex; gap:10px; align-items:center;">
    <input type="hidden" name="action" value="search">
```

“action = search” => triggers “searchStudent()” method

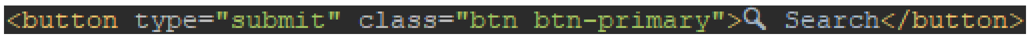
method= “get” => search appears in URL



```
<input type="text" name="keyword" placeholder="Search by code, name, email, or major"
       value="${keyword != null ? keyword: ''}"
       style="padding:10px; border-radius:5px; border:1px solid #ccc; flex:1;">
```

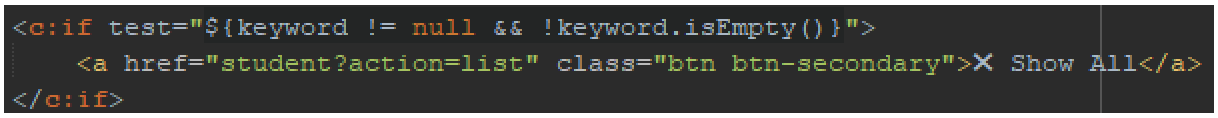
Whatever user types becomes “request.getParameter(“keyword”)”.

The “value” keeps the text after searching => ${keyword} is set by the controller.



```
<button type="submit" class="btn btn-primary">🔍 Search</button>
```

Submits the form => triggers search logic.



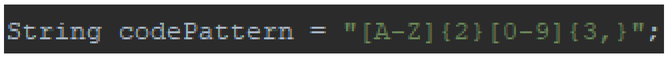
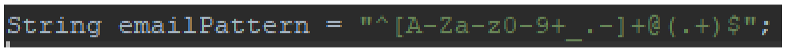
```
<c:if test="${keyword != null && !keyword.isEmpty()}">
    <a href="student?action=list" class="btn btn-secondary">✕ Show All</a>
</c:if>
```

Uses JSTL <c:if> to show this link “only if” a “keyword” exists.

Clicking resets the page by calling “student?action=list”.

Ex6.1:

private boolean validateStudent(Student student, HttpServletRequest request):

1. Student code
   
   ```
   String codePattern = "[A-Z]{2}[0-9]{3,}";
   ```
   - Must not be null or empty.
   - Must match 2 uppercase letters + ≥3 digits (e.g., SV001, IT123)
   - On failure: request.setAttribute("errorCode", "...")
2. Full Name
   - Must not be null or empty
   - Minimum 2 characters
   - On failure: request.setAttribute("errorName", "...")
3. Email
   
   ```
   String emailPattern = "^[A-Za-z0-9+_.-]+@(.+)$";
   ```
   - Only checked if not empty
   - Must match basic email format
   - On failure: request.setAttribute("errorEmail", "...")
4. Major
   - Must not be null or empty
   - On failure: request.setAttribute("errorMajor", "...")

Workflow:

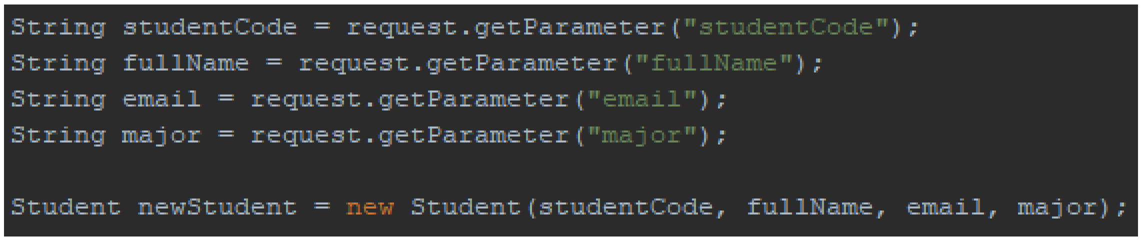
1. Start "isValid = true"
2. Check each field
3. If a field fails => set error message in request + "isValid = false"
4. Return "isValid"

   This allows the controller to:

- Forward back to the form if "false"
- Preserve user input and show specific field errors

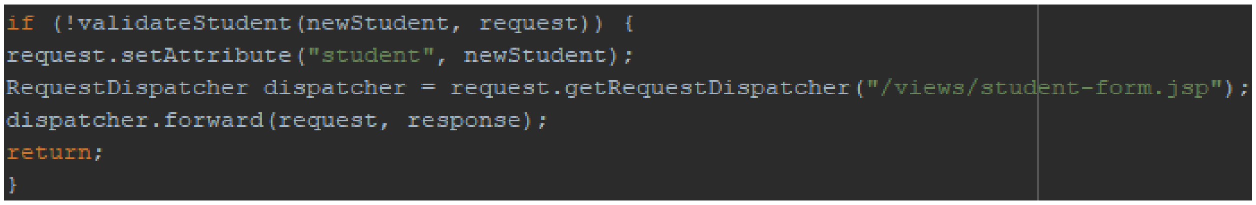
Ex6.2:

private void insertStudent(HttpServletRequest request, HttpServletResponse response):



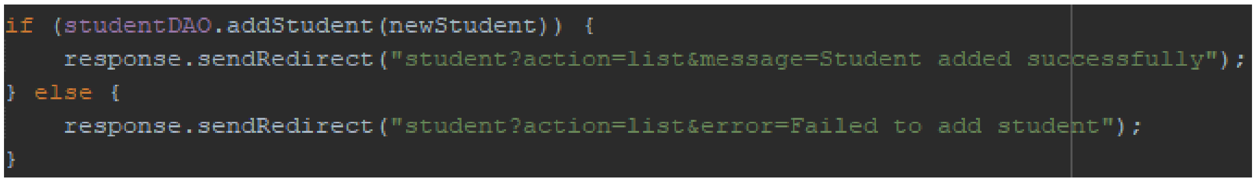
```java
String studentCode = request.getParameter("studentCode");
String fullName = request.getParameter("fullName");
String email = request.getParameter("email");
String major = request.getParameter("major");

Student newStudent = new Student(studentCode, fullName, email, major);
```

Reads input from the form and creates a new “Student” object.



```java
if (!validateStudent(newStudent, request)) {
request.setAttribute("student", newStudent);
RequestDispatcher dispatcher = request.getRequestDispatcher("/views/student-form.jsp");
dispatcher.forward(request, response);
return;
}
```

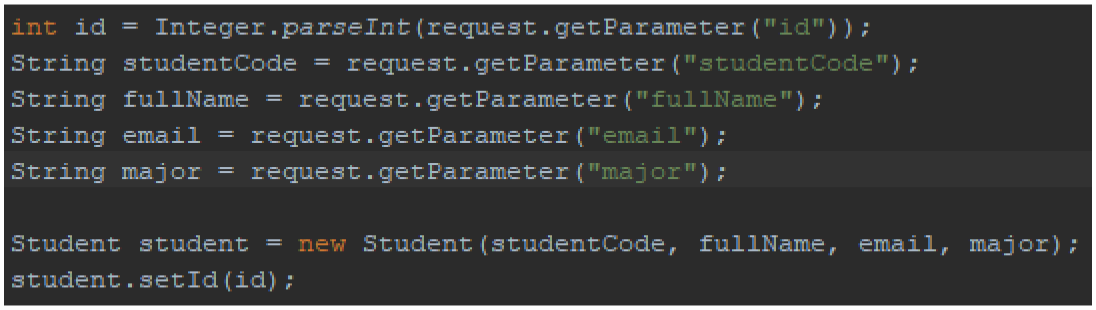
- Calls “validateStudent()”
- If any validation fails:
    - Set the student object as a request attribute (request.setAttribute("student", newStudent)) so the form keeps user input.
    - Forward back to the JSP form (dispatcher.forward(...))
    - Stops execution immediately with return.



```java
if (studentDAO.addStudent(newStudent)) {
    response.sendRedirect("student?action=list&message=Student added successfully");
} else {
    response.sendRedirect("student?action=list&error=Failed to add student");
}
```

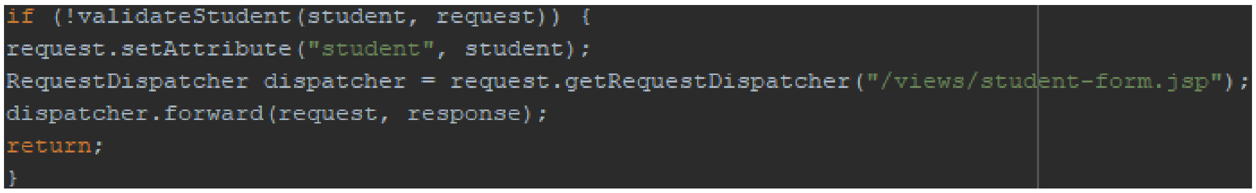
- If validation passed, calls DAO to insert into the database.
- Redirects to the list page with success or error message depending on DAO result.

private void updateStudent(HttpServletRequest request, HttpServletResponse response):



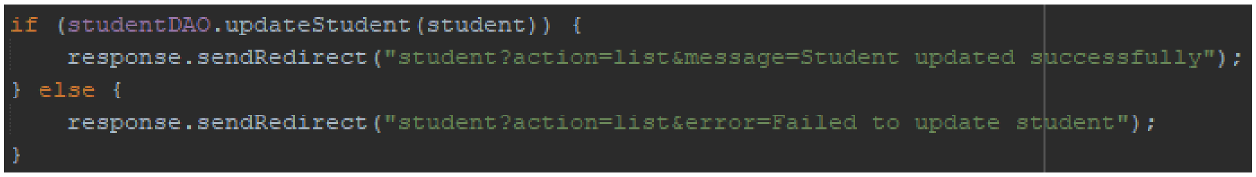
```java
int id = Integer.parseInt(request.getParameter("id"));
String studentCode = request.getParameter("studentCode");
String fullName = request.getParameter("fullName");
String email = request.getParameter("email");
String major = request.getParameter("major");

Student student = new Student(studentCode, fullName, email, major);
student.setId(id);
```

- Reads input from the form including the hidden “id”.
- Creates a “Student” object and sets its “id”.



```java
if (!validateStudent(student, request)) {
request.setAttribute("student", student);
RequestDispatcher dispatcher = request.getRequestDispatcher("/views/student-form.jsp");
dispatcher.forward(request, response);
return;
}
```

- Validates the student.
- On failure, preserves data and returns to the form, exactly like "insertStudent()".

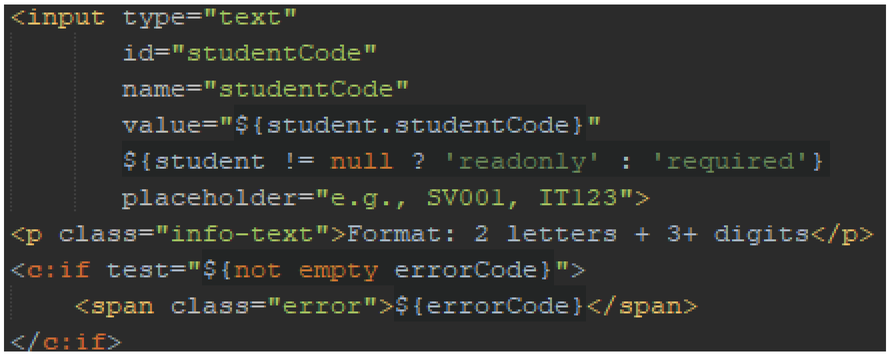


```java
if (studentDAO.updateStudent(student)) {
    response.sendRedirect("student?action=list&message=Student updated successfully");
} else {
    response.sendRedirect("student?action=list&error=Failed to update student");
}
```

- Calls DAO to update the database.
- Redirects to list page with feedback message.

Ex6.3:

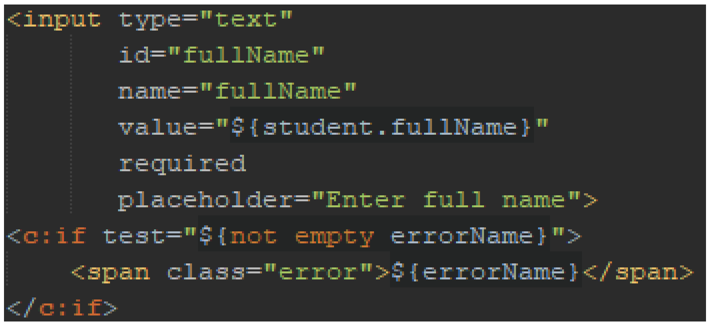
1. Student code



```jsp
<input type="text"
       id="studentCode"
       name="studentCode"
       value="${student.studentCode}"
       ${student != null ? 'readonly' : 'required'}
       placeholder="e.g., SV001, IT123">
<p class="info-text">Format: 2 letters + 3+ digits</p>
<c:if test="${not empty errorCode}">
    <span class="error">${errorCode}</span>
</c:if>
```

   - Shows the existing value (${student.studentCode}) if editing.
   - Makes the field **readonly** for edit mode (student != null) and required for new insert.
   - Displays error message from "validateStudent()" if "errorCode" exists.

2. Full Name



```jsp
<input type="text"
       id="fullName"
       name="fullName"
       value="${student.fullName}"
       required
       placeholder="Enter full name">
<c:if test="${not empty errorName}">
    <span class="error">${errorName}</span>
</c:if>
```

- Shows entered or existing value.
- Error message appears if “validateStudent()” sets “errorName”.

3. Email

```
<input type="email"
       id="email"
       name="email"
       value="${student.email}"
       required
       placeholder="student@example.com">
<c:if test="${not empty errorEmail}">
    <span class="error">${errorEmail}</span>
</c:if>
```

- Pre-fills the email field.
- Displays error if the email format is invalid (errorEmail).

4. Major

```
<select id="major" name="major" required>
    <option value="">-- Select Major --</option>
    <option value="Computer Science"
            ${student.major == 'Computer Science' ? 'selected' : ''}>
        Computer Science
    </option>
    <option value="Information Technology"
            ${student.major == 'Information Technology' ? 'selected' : ''}>
        Information Technology
    </option>
    <option value="Software Engineering"
            ${student.major == 'Software Engineering' ? 'selected' : ''}>
        Software Engineering
    </option>
    <option value="Business Administration"
            ${student.major == 'Business Administration' ? 'selected' : ''}>
        Business Administration
    </option>
</select>
            <c:if test="${not empty errorMajor}">
                <span class="error">${errorMajor}</span>
            </c:if>
```

- Preserves the previously selected major.
- Shows an error message if validation fails “errorMajor”.

Ex7.1:

1. public List<Student> getStudentsSorted(String sortBy, String order):

   - Returns a list of students sorted by the specified column “sortBy” and direction

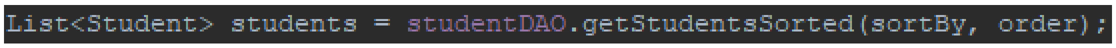
"order".
- Validates parameters:
+ "sortBy" must be one of: "id", "student_code", "full_name", "email", "major".
+ "order" must be "asc" or "desc".
+ Defaults: sortBy = "id", order = "asc" if invalid.
- Executes query, maps "ResultSet" to "Student" objects, returns the list.

2. public List<Student> getStudentsByMajor(String major):
   - Returns all students in a specific major.
   - Uses PreparedStatement to prevent SQL injection.
   - Sets the major parameter, executes query, maps "ResultSet" to "Student" objects.

Ex7.2:

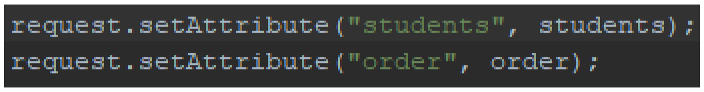
private void sortStudents(HttpServletRequest request, HttpServletResponse response):

- request.getParameter("sortBy") => reads the column to sort by.
- request.getParameter("order") => reads ascending or descending order.
- Calls DAO method:



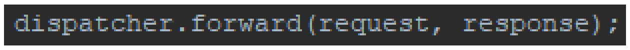
```
List<Student> students = studentDAO.getStudentsSorted(sortBy, order);
```

- Sets attributes for JSP to render the sorted table:



```
request.setAttribute("students", students);
request.setAttribute("order", order);
```

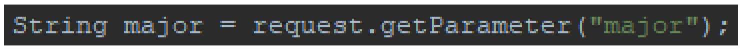
- Forwards to the JSP:



```
dispatcher.forward(request, response);
```

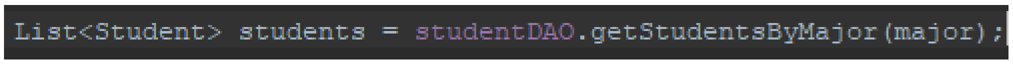
private void filterStudents(HttpServletRequest request, HttpServletResponse response):

- Reads the selected major:



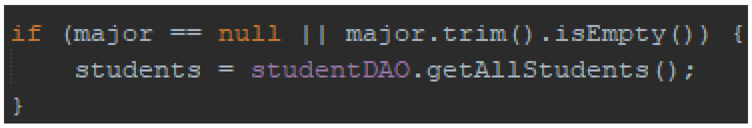
```
String major = request.getParameter("major");
```

- Calls DAO:



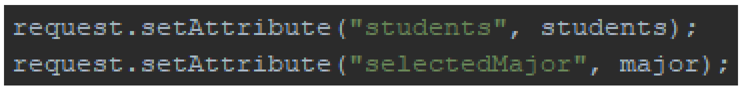
```
List<Student> students = studentDAO.getStudentsByMajor(major);
```

- Handles edge case for "All Majors" (empty selection):



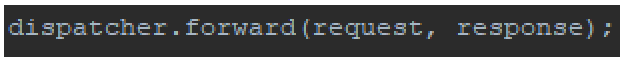
```
if (major == null || major.trim().isEmpty()) {
    students = studentDAO.getAllStudents();
}
```

- Sets attributes for JSP:



```
request.setAttribute("students", students);
request.setAttribute("selectedMajor", major);
```

- Forwards to the JSP to render the filtered table:



```
dispatcher.forward(request, response);
```

Ex8.1:

public int getTotalStudents():

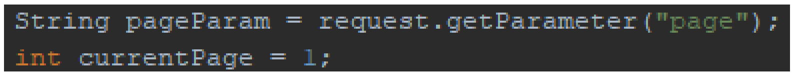
- Purpose: Count all student records in the database.
- SQL: SELECT COUNT(*) FROM students
- Logic:
  + Executes the query.
  + Uses “rs.getInt(1)” to retrieve the first column of the result (the count).
  + Returns the total number of students.
- If an exception occurs, it prints the stack trace and returns 0.
- Use case: Needed to calculate the total number of pages for pagination.

public List<Student> getStudentsPaginated(int offset, int limit):

- Purpose: Fetch a subset of students for a specific page.
- SQL: SELECT * FROM students ORDER BY id DESC LIMIT ? OFFSET ?
  + LIMIT => number of records per page.
  + OFFSET => starting position of records (calculated from current page).
- Logic:
  + Sets the “limit” and “offset” parameters in the “PreparedStatement”.
  + Executes the query.
  + Loops through the “ResultSet” to populate “Student” objects.
  + Returns a “List<Student>” containing only the students for that page.
- Use case: Used by the controller to show paginated results and to display “Next”/“Previous” pages.

Ex8.2:

private void listStudents(HttpServletRequest request, HttpServletResponse response):



```
String pageParam = request.getParameter("page");
int currentPage = 1;
```

- Default to page 1.
- Try parsing “pageParam”. If invalid (e.g., non-numeric), fall back to 1.

```java
int recordsPerPage = 10;
```

- Each page will show 10 students.

```java
int totalRecords = studentDAO.getTotalStudents();
int totalPages = (int) Math.ceil((double) totalRecords / recordsPerPage);
```

- “getTotalStudents()” counts all students in the DB.
- “totalPages” = ceiling of total records ÷ records per page.

```java
if (currentPage < 1) {
    currentPage = 1;
}
if (currentPage > totalPages && totalPages > 0) {
    currentPage = totalPages;
}
```

- Ensures “currentPage” is never less than 1 or greater than the total number of pages.

```java
int offset = (currentPage - 1) * recordsPerPage;
```

- Determines starting row for the current page.

```java
List<Student> students = studentDAO.getStudentsPaginated(offset, recordsPerPage);
```

- Only retrieves students for the current page using “LIMIT” and “OFFSET”.

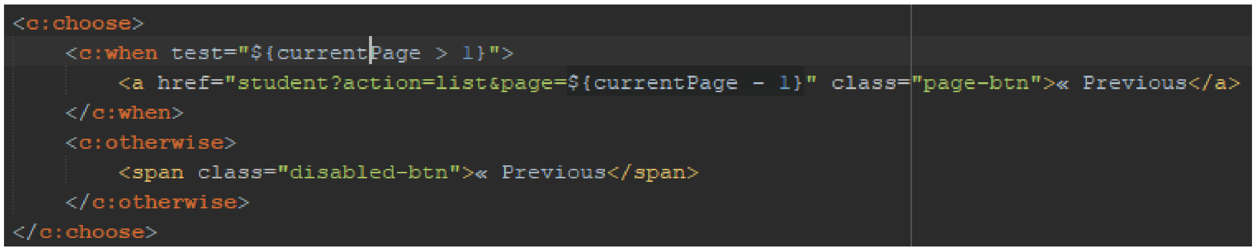
```java
request.setAttribute("students", students);
request.setAttribute("currentPage", currentPage);
request.setAttribute("totalPages", totalPages);
request.setAttribute("recordsPerPage", recordsPerPage);
request.setAttribute("totalRecords", totalRecords);
```

- Sends student list + pagination info to “student-list.jsp”.

```java
RequestDispatcher dispatcher = request.getRequestDispatcher("/views/student-list.jsp");
dispatcher.forward(request, response);
```

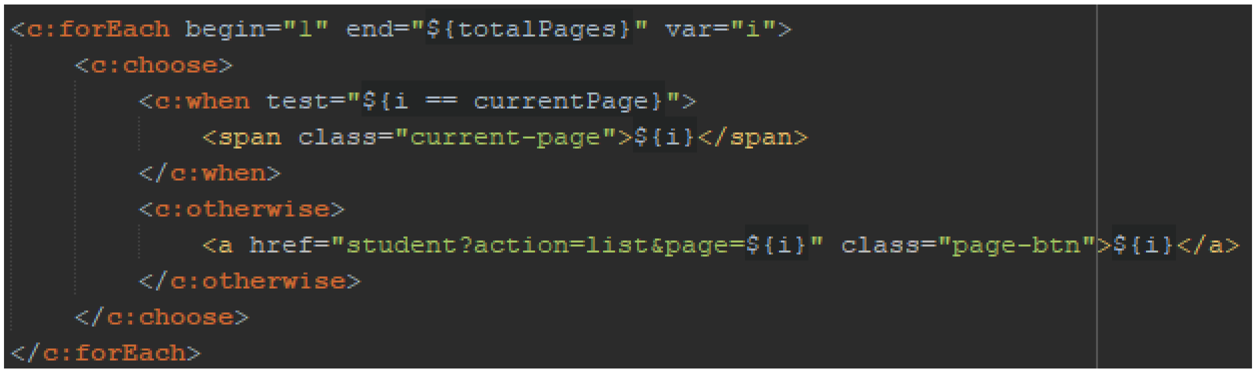
- The JSP uses these attributes to display the table and pagination UI.

Ex8.3:



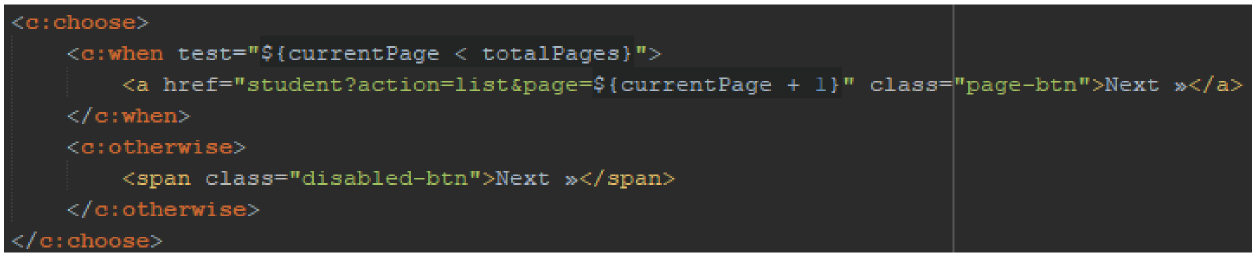
```
<c:choose>
    <c:when test="${currentPage > 1}">
        <a href="student?action=list&page=${currentPage - 1}" class="page-btn">« Previous</a>
    </c:when>
    <c:otherwise>
        <span class="disabled-btn">« Previous</span>
    </c:otherwise>
</c:choose>
```

- Enabled only if “currentPage” > 1.
- Otherwise, shows a disabled span.



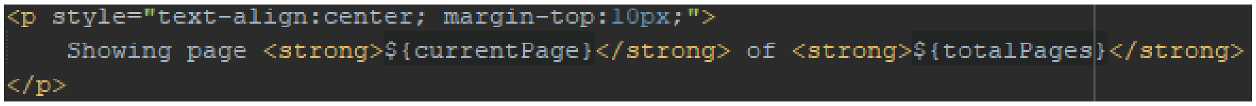
```
<c:forEach begin="1" end="${totalPages}" var="i">
    <c:choose>
        <c:when test="${i == currentPage}">
            <span class="current-page">${i}</span>
        </c:when>
        <c:otherwise>
            <a href="student?action=list&page=${i}" class="page-btn">${i}</a>
        </c:otherwise>
    </c:choose>
</c:forEach>
```

- Loops from 1 to “totalPages”.
- Highlights the current page (<span class="current-page">) and makes others clickable links.



```
<c:choose>
    <c:when test="${currentPage < totalPages}">
        <a href="student?action=list&page=${currentPage + 1}" class="page-btn">Next »</a>
    </c:when>
    <c:otherwise>
        <span class="disabled-btn">Next »</span>
    </c:otherwise>
</c:choose>
```

- Enabled only if “currentPage” < “totalPages”.
- Disabled when on the last page.



```
<p style="text-align:center; margin-top:10px;">
    Showing page <strong>${currentPage}</strong> of <strong>${totalPages}</strong>
</p>
```

- Displays a simple "Showing page X of Y" message.