To create an Employee Management System using basic Java concepts and Object-Oriented Programming (OOP) principles, you can break down the project into key components like:

- Employee details (name, ID, designation, salary)
- Basic operations such as:
  - Add new employee
  - View employee details
  - Update employee details
  - Delete employee

Flow Chart for Employee Management System

Here's a simple flow:

1. **Start**
2. **Main Menu**
    5. Add Employee
    6. View Employee
    7. Update Employee
    8. Delete Employee
    9. Exit
3. **Perform the selected operation**
    - If 1: Go to **Add Employee**
    - If 2: Go to **View Employee**
    - If 3: Go to **Update Employee**
    - If 4: Go to **Delete Employee**
    - If 5: Exit the program
4. **End**

Code for Employee Management System

```java
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Scanner;

class Employee {
    private int id;
    private String name;
    private String designation;
    private double salary;
    private String department;

    // Constructor
    public Employee(int id, String name, String designation, double salary, String department) {
        this.id = id;
        this.name = name;
        this.designation = designation;
        this.salary = salary;
        this.department = department;
    }

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDesignation() {
        return designation;
    }

    public void setDesignation(String designation) {
        this.designation = designation;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public String getDepartment() {
        return department;
    }

    public void setDepartment(String department) {
        this.department = department;
    }

    // Display employee details
    public void displayEmployee() {
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Designation: " + designation);
        System.out.println("Salary: " + salary);
        System.out.println("Department: " + department);
        System.out.println("------------------------------");
    }
}
//class to perform all functionalities
class EmployeeManagementSystem {
    private ArrayList<Employee> employees = new ArrayList<>();
    private Scanner scanner = new Scanner(System.in);

    // Add new employee
    public void addEmployee() {
        System.out.println("Enter Employee ID:");
        int id = scanner.nextInt();
        scanner.nextLine(); // consume newline
        System.out.println("Enter Employee Name:");
        String name = scanner.nextLine();
        System.out.println("Enter Employee Designation:");
        String designation = scanner.nextLine();
        System.out.println("Enter Employee Salary:");
        double salary = scanner.nextDouble();
        scanner.nextLine(); // consume newline
        System.out.println("Enter Employee Department:");
        String department = scanner.nextLine();

        Employee employee = new Employee(id, name, designation, salary, department);
        employees.add(employee);
        System.out.println("Employee added successfully!");
    }

    // View all employees
    public void viewEmployees() {
        if (employees.isEmpty()) {
            System.out.println("No employees to display.");
        } else {
            for (Employee employee : employees) {
                employee.displayEmployee();
            }
        }
    }

    // Update employee details
    public void updateEmployee() {
        System.out.println("Enter Employee ID to update:");
        int id = scanner.nextInt();
        scanner.nextLine(); // consume newline
        boolean found = false;

        for (Employee employee : employees) {
            if (employee.getId() == id) {
                found = true;
                System.out.println("Enter new Employee Name:");
                String name = scanner.nextLine();
                System.out.println("Enter new Employee Designation:");
                String designation = scanner.nextLine();
                System.out.println("Enter new Employee Salary:");
                double salary = scanner.nextDouble();
                scanner.nextLine(); // consume newline
                System.out.println("Enter new Employee Department:");
                String department = scanner.nextLine();

                employee.setName(name);
                employee.setDesignation(designation);
                employee.setSalary(salary);
                employee.setDepartment(department);

                System.out.println("Employee updated successfully!");
                break;
            }
        }

        if (!found) {
            System.out.println("Employee with ID " + id + " not found.");
        }
    }

    // Delete an employee
    public void deleteEmployee() {
        System.out.println("Enter Employee ID to delete:");
        int id = scanner.nextInt();
        boolean found = false;

        for (Employee employee : employees) {
            if (employee.getId() == id) {
                found = true;
                employees.remove(employee);
                System.out.println("Employee deleted successfully!");
                break;
            }
        }

        if (!found) {
            System.out.println("Employee with ID " + id + " not found.");
        }
    }

    // Sort employees by department and display them
    public void sortByDepartment() {
        if (employees.isEmpty()) {
            System.out.println("No employees to display.");
            return;
        }

        System.out.println("Enter the department to sort by:");
        String department = scanner.nextLine();
        ArrayList<Employee> sameDepartmentEmployees = new ArrayList<>();

        // Collect employees who are in the same department
```

```
        for (Employee employee : employees) {
            if (employee.getDepartment().equalsIgnoreCase(department)) {
                sameDepartmentEmployees.add(employee);
            }
        }

        // Sort employees by name within the department
        sameDepartmentEmployees.sort(Comparator.comparing(Employee::getName));

        // Display sorted employees
        if (sameDepartmentEmployees.isEmpty()) {
            System.out.println("No employees found in department: " + department);
        } else {
            System.out.println("Employees in department: " + department);
            for (Employee employee : sameDepartmentEmployees) {
                employee.displayEmployee();
            }
        }
    }

    // Menu to perform operations
    public void showMenu() {
        int choice;

        do {
            System.out.println("\n*** Employee Management System ***");
            System.out.println("1. Add Employee");
            System.out.println("2. View Employees");
            System.out.println("3. Update Employee");
            System.out.println("4. Delete Employee");
            System.out.println("5. Sort Employees by Department");
            System.out.println("6. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();
            scanner.nextLine(); // consume newline

            switch (choice) {
                case 1:
                    addEmployee();
                    break;
                case 2:
                    viewEmployees();
                    break;
                case 3:
                    updateEmployee();
                    break;
                case 4:
                    deleteEmployee();
                    break;
                case 5:
                    sortByDepartment();
                    break;
                case 6:
                    System.out.println("Exiting the system.");
                    break;
                default:
                    System.out.println("Invalid choice! Please try again.");
            }
        } while (choice != 6);
    }
}

public class Main {
    public static void main(String[] args) {
        EmployeeManagementSystem ems = new EmployeeManagementSystem();
        ems.showMenu();
    }
}
```

Explanation of the Code

1. **Employee Class:**
    - This class models an employee with basic attributes: `id`, `name`, `designation`, and `salary`.
    - It provides a constructor to initialize employee objects and getter/setter methods to access and modify employee attributes.
    - The `displayEmployee` method is used to print employee details.
2. **EmployeeManagementSystem Class:**
    - This class handles the core functionalities of the system. It maintains a list of employees using an `ArrayList`.
    - It has methods for adding, viewing, updating, and deleting employees:
        - **addEmployee:** Takes user input and creates an employee.
        - **viewEmployees:** Iterates through the list and prints employee details.
        - **updateEmployee:** Updates the information of an employee based on the ID.
        - **deleteEmployee:** Removes an employee from the list by matching the ID.
    - **showMenu:** Displays the main menu and prompts the user for input. Based on the input, it calls appropriate methods.
3. **Main Class:**
    - The `Main` class contains the `main` method, which initializes the `EmployeeManagementSystem` object and calls `showMenu` to start the interaction.

This implementation provides a simple console-based application that covers basic CRUD (Create, Read, Update, Delete) operations on employees.

Explanation of the Changes

1. **Added Department Attribute:**
    - In the `Employee` class, I've added a `department` attribute to store the department each employee belongs to.
    - Updated the constructor and getter/setter methods to handle this new attribute.
2. **addEmployee Method:**
    - When adding an employee, the program now prompts the user to input the employee's department along with the other details like ID, name, designation, and salary.
3. **updateEmployee Method:**
    - The `updateEmployee` method now allows updating the department of an employee in addition to the other attributes.
4. **sortByDepartment Method:**

    - This is a new method added to sort and display employees who belong to the same department.
    - It first collects all employees from the user-specified department, then sorts them alphabetically by their name using the `Comparator.comparing` method.
    - If no employees are found in the specified department, a message is displayed.
5. **Menu Update:**
    - Updated the main menu to include a new option (5) to sort employees by department, and shifted the exit option to (6).

This updated program allows you to manage employees by their departments and also provides a sorting feature to list employees within the same department.

Explanation of Code line by line

Imports

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Scanner;
```

- `ArrayList`: Allows us to store and manage dynamic collections of employees.
- `Comparator`: Enables sorting based on specific attributes (like name) in the program.
- `Scanner`: Allows reading user input from the console.

Employee Class Definition

```
class Employee {
    private int id;
    private String name;
    private String designation;
    private double salary;
    private String department;
```

- **Attributes**: `id`, `name`, `designation`, `salary`, and `department` are private fields representing the employee's details.

Constructor and Methods in `Employee`

```
   public Employee(int id, String name, String designation, double
salary, String department) {
        this.id = id;
        this.name = name;
        this.designation = designation;
        this.salary = salary;
        this.department = department;
    }
```

- **Constructor**: Initializes the fields of the `Employee` object when created. When a new employee is added, this constructor is called with the input values.

Getters And Setters function

```
public int getId() { return id; }

    public void setId(int id) { this.id = id; }


    public String getName() { return name; }

    public void setName(String name) { this.name = name; }


    public String getDesignation() { return designation; }

    public void setDesignation(String designation) { this.designation =
designation; }


    public double getSalary() { return salary; }

    public void setSalary(double salary) { this.salary = salary; }


    public String getDepartment() { return department; }
    public void setDepartment(String department) { this.department =
department; }
```

- **Getters and Setters: These are public methods to access or modify private fields (`id`, `name`, `designation`, etc.) outside the class.**

1. Getter and Setter for `id`

```
public int getId() { return id; }
public void setId(int id) { this.id = id; }
```

- `getId()`:
  - This is a **getter method**. It allows you to access (or "get") the private field `id` from outside the `Employee` class.
  - Since `id` is private (i.e., it can only be accessed within the `Employee` class), the getter method provides a way to retrieve its value.
  - It returns the value of the `id` field.
- `setId(int id)`:
  - This is a **setter method**. It allows you to modify (or "set") the private field `id` from outside the `Employee` class.
  - It takes an `int` parameter (`id`) and assigns this value to the class's private `id` field.
  - `this.id = id`: Here, `this.id` refers to the current object's `id` field, and the `id` on the right side refers to the parameter passed to the method.

2. Getter and Setter for `name`

```
public String getName() { return name; }
public void setName(String name) { this.name = name; }
```

- `getName()`:
  - This is a getter method for the `name` field, which returns the employee's name (a `String`).
  - It provides access to the `name` field from outside the class.
- `setName(String name)`:

    - This is a setter method for the `name` field. It takes a `String` parameter (`name`) and assigns it to the employee's `name` field.
    - `this.name = name`: Here, `this.name` refers to the current object's `name` field, and the `name` on the right side refers to the method's parameter.

3. Getter and Setter for `designation`

```
public String getDesignation() { return designation; }
public void setDesignation(String designation) { this.designation =
designation; }
```

- **`getDesignation()`**:
    - This getter method returns the value of the private `designation` field.
    - It allows code outside the `Employee` class to access the employee's designation.
- **`setDesignation(String designation)`**:
    - This setter method takes a `String` parameter (`designation`) and assigns it to the `designation` field of the employee.
    - `this.designation = designation`: This sets the value of the current object's `designation` field to the value passed as an argument.

4. Getter and Setter for `salary`

```
public double getSalary() { return salary; }
public void setSalary(double salary) { this.salary = salary; }
```

- **`getSalary()`**:
    - This getter method returns the value of the employee's `salary` field, which is a `double`.
    - It allows external code to retrieve the salary of the employee.
- **`setSalary(double salary)`**:
    - This setter method takes a `double` parameter (`salary`) and sets the value of the employee's `salary` field to this value.
    - `this.salary = salary`: Assigns the argument `salary` to the current employee object's `salary` field.

5. Getter and Setter for `department`

```
public String getDepartment() { return department; }
public void setDepartment(String department) { this.department =
department; }
```

- **`getDepartment()`**:
  - This getter method returns the value of the employee's `department` field (a `String`).
  - It provides external access to the `department` field.
- **`setDepartment(String department)`**:
  - This setter method takes a `String` parameter (`department`) and sets the employee's `department` field to this value.
  - `this.department = department`: This sets the current object's `department` field to the value passed as an argument.

Summary:

- **Getters** (`getId()`, `getName()`, `getDesignation()`, `getSalary()`, `getDepartment()`):
  - These methods are used to **retrieve** the values of private fields. They return the current value of a specific field of the `Employee` object.
- **Setters** (`setId(int id)`, `setName(String name)`, `setDesignation(String designation)`, `setSalary(double salary)`, `setDepartment(String department)`):
  - These methods are used to **modify** the values of private fields. They take a parameter and assign it to the corresponding field of the `Employee` object.

Why Getters and Setters?

- **Encapsulation**: In object-oriented programming, one of the key principles is **encapsulation**. By making fields private and using getters and setters, we control how the fields of an object are accessed and modified. This hides the internal implementation and makes sure that fields are not modified in unintended ways.

For example, instead of directly accessing `id` or `name`, you use methods like `getId()` or `setName()`. This allows validation, logging, or other operations to be added in these methods later if needed, while still keeping the data secure.

`Display Employee Method in employee class`

```
public void displayEmployee() {
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Designation: " + designation);
        System.out.println("Salary: " + salary);
        System.out.println("Department: " + department);
        System.out.println("------------------------------");
    }
```

- **displayEmployee()**: Prints out the details of an employee (ID, name, designation, salary, and department).

EmployeeManagementSystem Class Definition

```
class EmployeeManagementSystem {
    private ArrayList<Employee> employees = new ArrayList<>();
    private Scanner scanner = new Scanner(System.in);
```

- **Attributes**:
  - `employees`: An `ArrayList` to store multiple `Employee` objects.
  - `scanner`: A `Scanner` object to take input from the user.

Adding an Employee

```
   public void addEmployee() {
        System.out.println("Enter Employee ID:");
        int id = scanner.nextInt();
        scanner.nextLine(); // consume newline
        System.out.println("Enter Employee Name:");
        String name = scanner.nextLine();
        System.out.println("Enter Employee Designation:");
        String designation = scanner.nextLine();
        System.out.println("Enter Employee Salary:");
        double salary = scanner.nextDouble();
        scanner.nextLine(); // consume newline
        System.out.println("Enter Employee Department:");
        String department = scanner.nextLine();

        Employee employee = new Employee(id, name, designation,
salary, department);
        employees.add(employee);
        System.out.println("Employee added successfully!");
    }
```

- **addEmployee()**: This method collects details from the user for a new employee (ID, name, designation, salary, and department) and creates a new `Employee` object. The new object is then added to the `employees` list.

Viewing Employees

```
public void viewEmployees() {
     if (employees.isEmpty()) {
         System.out.println("No employees to display.");
     } else {
         for (Employee employee : employees) {
             employee.displayEmployee();
         }
     }
 }
```

Purpose of the Method

The `viewEmployees()` method is designed to display the details of all employees stored in the `employees` list. If there are no employees, it informs the user that the list is empty.

Code Breakdown

```
public void viewEmployees() {
```

- **`public`**: This indicates that the method can be called from any other class (i.e., it is accessible publicly).
- **`void`**: This means the method does not return any value. Its purpose is to perform an action (displaying employee details) rather than return data.
- **`viewEmployees()`**: The method name, indicating that it will "view" or display employee information.

```
if (employees.isEmpty()) {
    System.out.println("No employees to display.");
}
```

- **`if (employees.isEmpty())`**:
  - This checks if the `employees` list is empty using the `isEmpty()` method, which returns `true` if the list contains no elements.
  - **`employees`** is an `ArrayList<Employee>` that stores all the employee objects.
- **`System.out.println("No employees to display.");`**:

    - If the `employees` list is empty (i.e., no employees have been added to the system), this line will print the message `"No employees to display."` to the console. This informs the user that there are no employees in the system.
    - **System.out.println()**: This is a standard Java method used to print text to the console.

```
else {
    for (Employee employee : employees) {
        employee.displayEmployee();
    }
}
```

- **`else`**: This block executes if the `employees` list is **not empty**.
    - If there are employees in the list, the code will proceed to the `else` block.
- **`for (Employee employee : employees)`**:
    - This is an **enhanced for loop** (also known as a "for-each" loop) that iterates through all the elements of the `employees` list.
    - For each iteration, an `Employee` object from the list is temporarily assigned to the variable `employee`.
    - The loop will continue until every `Employee` object in the list has been processed.
- **`employee.displayEmployee();`**:
    - This line calls the `displayEmployee()` method on each `Employee` object.
    - The `displayEmployee()` method (which is defined in the `Employee` class) prints all the details of the employee to the console (ID, name, designation, salary, department).
    - So, for each employee in the list, this method will print the employee's details.

How the Method Works

1. **Check if the list is empty**:
    - First, the method checks if the `employees` list contains any employees by using `isEmpty()`.
    - If the list is empty, it displays a message saying "No employees to display."
2. **Iterate through the employee list**:
    - If the list is not empty, it loops through all the `Employee` objects in the `employees` list.

3. **Display employee details**:
    - For each employee, it calls the `displayEmployee()` method, which prints the employee's details (such as ID, name, designation, salary, and department) to the console.

Summary

- The `viewEmployees()` method is a way to display all the employee data in the system.
- If no employees exist, the user is notified.
- If employees exist, their details are displayed one by one, using the `displayEmployee()` method for each `Employee` object in the list.

This method helps manage the employee information display without manually handling each employee's details individually.

Updating an employee

```
public void updateEmployee() {
        System.out.println("Enter Employee ID to update:");
        int id = scanner.nextInt();
        scanner.nextLine(); // consume newline
        boolean found = false;

        for (Employee employee : employees) {
            if (employee.getId() == id) {
                found = true;
                System.out.println("Enter new Employee Name:");
                String name = scanner.nextLine();
                System.out.println("Enter new Employee Designation:");
                String designation = scanner.nextLine();
                System.out.println("Enter new Employee Salary:");
                double salary = scanner.nextDouble();
                scanner.nextLine(); // consume newline
                System.out.println("Enter new Employee Department:");
                String department = scanner.nextLine();

                employee.setName(name);
                employee.setDesignation(designation);
                employee.setSalary(salary);
                employee.setDepartment(department);
                System.out.println("Employee updated successfully!");
                break;
            }
        }

        if (!found) {
            System.out.println("Employee with ID " + id + " not 
found.");
        }
    }
```

Let’s break down the `updateEmployee()` method in detail, explaining what each part does and how the logic flows.

Overview

The `updateEmployee()` method allows the user to update the details (name, designation, salary, department) of an employee based on the provided employee ID. The method interacts with the user through the console to gather the new details for the employee and updates the corresponding employee’s information.

Code Breakdown

```
public void updateEmployee() {
```

- **`public`**: This means the method can be called from anywhere.
- **`void`**: This means the method does not return any value; it performs an action.
- **`updateEmployee()`**: The method name, indicating that it is responsible for updating an employee’s details.

```
System.out.println("Enter Employee ID to update:");
```

```
int id = scanner.nextInt();

scanner.nextLine(); // consume newline
```

- **`System.out.println("Enter Employee ID to update:");`**:
    - This prints a prompt message to the console asking the user to enter the ID of the employee whose details they want to update.
- **`int id = scanner.nextInt();`**:
    - The `scanner.nextInt()` reads the employee ID (an integer) from the console input and stores it in the variable `id`. This ID will be used to search for the corresponding employee.
- **`scanner.nextLine();`**:
    - This is used to consume the newline character left by `nextInt()`. Without this line, the next `scanner.nextLine()` call would be skipped, causing issues with reading input properly.

```
boolean found = false;
```

- A **boolean flag** named `found` is initialized to `false`.
- This flag is used to track whether an employee with the given ID is found during the search in the employee list.
- If the employee is found, this flag will be set to `true`.

```
for (Employee employee : employees) {
```

- This is a **for-each loop** that iterates over the `employees` list.
- For each iteration, an `Employee` object from the `employees` list is temporarily assigned to the variable `employee`.
- The loop continues until every employee in the list has been processed.

```
if (employee.getId() == id) {

    found = true;
```

- Inside the loop, we check if the current employee's `id` matches the `id` entered by the user.
- **`employee.getId() == id`**: The `getId()` method retrieves the `id` of the current `Employee` object and compares it with the input `id`.
- If the employee with the matching ID is found, we set `found = true` to indicate that the employee was located, and we proceed to update the employee's details.

```
System.out.println("Enter new Employee Name:");

String name = scanner.nextLine();
```

- Once an employee with the matching ID is found, the system prompts the user to enter the new name for the employee.
- **`String name = scanner.nextLine();`**: This reads the new name entered by the user as a string and stores it in the variable `name`.

```
System.out.println("Enter new Employee Designation:");

String designation = scanner.nextLine();
```

- The system prompts the user to enter the new designation for the employee.
- **`String designation = scanner.nextLine();`**: This reads the new designation entered by the user as a string and stores it in the variable `designation`.

```
System.out.println("Enter new Employee Salary:");
```

```java
double salary = scanner.nextDouble();

scanner.nextLine(); // consume newline
```

- The system prompts the user to enter the new salary for the employee.
- **`double salary = scanner.nextDouble();`**: This reads the new salary (a `double` value) entered by the user and stores it in the variable `salary`.
- The `scanner.nextLine();` again consumes the newline character left by `nextDouble()` to ensure proper input reading for the next line.

```java
System.out.println("Enter new Employee Department:");

String department = scanner.nextLine();
```

- The system prompts the user to enter the new department for the employee.
- **`String department = scanner.nextLine();`**: This reads the new department as a string and stores it in the variable `department`.

```java
employee.setName(name);

employee.setDesignation(designation);

employee.setSalary(salary);

employee.setDepartment(department);
```

- Now that we have the new details for the employee, we use the **setter methods** of the `Employee` class to update the employee's details.
  - **`employee.setName(name);`**: This updates the employee's name with the new value entered by the user.

- ◦ **`employee.setDesignation(designation);`**: This updates the employee's designation with the new value entered by the user.
  - ◦ **`employee.setSalary(salary);`**: This updates the employee's salary with the new value entered by the user.
  - ◦ **`employee.setDepartment(department);`**: This updates the employee's department with the new value entered by the user.

```
System.out.println("Employee updated successfully!");

break;
```

- After updating the employee's details, the message `"Employee updated successfully!"` is printed to confirm the update.
- **`break;`**: This exits the `for` loop after the employee is found and updated. There is no need to continue the loop once the matching employee has been updated.

```
if (!found) {

    System.out.println("Employee with ID " + id + " not found.");

}
```

- After the loop has completed, we check if the `found` flag is still `false`.
  - If `found` is `false`, it means no employee with the given ID was found in the list.
  - If that's the case, the system prints a message like `"Employee with ID 101 not found."`, where `101` is the entered ID.

How the Method Works

1. **Prompt for Employee ID**: The user is asked to input the ID of the employee they want to update.
2. **Search for Employee**: The method loops through the `employees` list and checks each employee's ID.
3. **Update Employee Details**:
   - If an employee with the entered ID is found, the system prompts the user to input the updated details (name, designation, salary, and department).
   - The employee's details are then updated using the setter methods.
4. **Confirmation**: If the employee was found and updated, a success message is displayed. If no employee with the given ID is found, the user is informed.

Summary

This method allows for the interactive update of an employee's details based on the ID. It performs a search using the ID, takes new details from the user, and updates the corresponding fields of the employee object. If the ID does not exist in the list, it informs the user that no such employee was found.

Delete an Employee

```
  public void deleteEmployee() {

    System.out.println("Enter Employee ID to delete:");

    int id = scanner.nextInt();

    boolean found = false;


    for (Employee employee : employees) {

      if (employee.getId() == id) {

        found = true;

        employees.remove(employee);

        System.out.println("Employee deleted successfully!");

        break;

      }

    }
```

```
        if (!found) {
            System.out.println("Employee with ID " + id not found.");
        }
    }
```

Let's go through the `deleteEmployee()` method step by step, explaining each part in detail.

Overview

The `deleteEmployee()` method allows the user to remove an employee from the `employees` list by entering the employee's ID. It searches for the employee by ID, removes the employee if found, and informs the user whether the deletion was successful or if the employee was not found.

Code Breakdown

```
public void deleteEmployee() {
```

- **`public`**: This means the method can be accessed from any other class.
- **`void`**: The method does not return any value; it performs an action (deleting an employee).
- **`deleteEmployee()`**: The method name, indicating that it will handle the deletion of an employee.

```
System.out.println("Enter Employee ID to delete:");

int id = scanner.nextInt();
```

- **`System.out.println("Enter Employee ID to delete:");`**: This line prints a message to the console asking the user to input the ID of the employee they wish to delete.

- **`int id = scanner.nextInt();`**: The program waits for the user to enter an employee ID. The `scanner.nextInt()` method reads the input (an integer) from the console and stores it in the variable `id`.

```
boolean found = false;
```

- A **boolean flag** named `found` is initialized to `false`. This variable is used to track whether an employee with the entered ID is found during the search.
- If the employee is found, this flag will be set to `true`. If not, it remains `false`, indicating that no employee with the entered ID was found.

```
for (Employee employee : employees) {
```

- This is a **for-each loop** that iterates through the `employees` list.
- For each iteration, an `Employee` object from the `employees` list is temporarily assigned to the variable `employee`.
- The loop will continue until every employee in the list has been checked.

```
if (employee.getId() == id) {

    found = true;
```

- **`if (employee.getId() == id)`**: This condition checks if the current `Employee` object's `id` matches the ID entered by the user.
  - The `getId()` method retrieves the employee's `id` and compares it with the `id` entered by the user.
  - If a match is found, the condition evaluates to `true`.

- **`found = true;`**: Once an employee with the matching ID is found, we set the `found` flag to `true` to indicate that the employee has been located.

```
employees.remove(employee);
```

- **`employees.remove(employee);`**: This line removes the `employee` object from the `employees` list.
    - The `ArrayList` class's `remove()` method is used to delete the employee.
    - Once an employee is removed, it is no longer stored in the list and cannot be accessed later.

```
System.out.println("Employee deleted successfully!");

break;
```

- **`System.out.println("Employee deleted successfully!");`**: After successfully removing the employee, this message is printed to inform the user that the employee has been deleted.
- **`break;`**: The `break` statement is used to exit the `for` loop early. Once the employee has been found and deleted, there's no need to continue looping through the list, so the `break` stops further iterations.

```
if (!found) {

    System.out.println("Employee with ID " + id + " not found.");

}
```

- **`if (!found)`**: This checks if the `found` flag is still `false` after the loop has finished.
    - If no employee with the entered ID was found, the `found` flag remains `false`.
- **`System.out.println("Employee with ID " + id + " not found.");`**:
    - If no employee with the specified ID was found, this line prints a message like `"Employee with ID 101 not found."`, where `101` is the entered ID.
    - This informs the user that no employee with the entered ID exists in the system.

How the Method Works

1. **Prompt for Employee ID**: The user is asked to input the ID of the employee they wish to delete.
2. **Search for Employee**: The method loops through the `employees` list, checking each employee's ID.
3. **Delete the Employee**:
    - If an employee with the entered ID is found, the employee is removed from the list, and a success message is printed.
4. **Check if Employee Exists**:
    - If no employee with the entered ID is found, the method prints a message informing the user that the employee does not exist.

Summary

- **Input**: The user enters the ID of the employee to be deleted.
- **Search and Delete**: The method searches the list of employees to find the one with the matching ID. If found, the employee is deleted.
- **Confirmation**: The method provides feedback to the user, confirming either the successful deletion of the employee or informing them that no employee with the specified ID was found.

This method ensures efficient deletion and provides appropriate feedback based on the user's input.

Sort Employee by department

```
public void sortByDepartment() {
    if (employees.isEmpty()) {
        System.out.println("No employees to display.");
        return;
    }

    System.out.println("Enter the department to sort by:");
    String department = scanner.nextLine();
    ArrayList<Employee> sameDepartmentEmployees = new ArrayList<>();

    for (Employee employee : employees) {
        if (employee.getDepartment().equalsIgnoreCase(department)) {
            sameDepartmentEmployees.add(employee);
        }
    }

    sameDepartmentEmployees.sort(Comparator.comparing(Employee::getName));

    if (sameDepartmentEmployees.isEmpty()) {
        System.out.println("No employees found in department: " + department);
    } else {
        System.out.println("Employees in department: " + department);
        for (Employee employee : sameDepartmentEmployees) {
            employee.displayEmployee();
        }
      }
    }
```

Let's walk through the `sortByDepartment()` method line by line and explain what each part does in detail.

Overview

The `sortByDepartment()` method allows the user to filter and sort employees by their department. It first asks the user to input a department name, finds employees in that department, sorts them by name, and then displays the sorted list. If no employees are found in the specified department, it informs the user accordingly.

Code Breakdown

```
public void sortByDepartment() {
```

- **`public`**: The method can be accessed from outside this class, in this case from the `showMenu()` method.
- **`void`**: The method does not return any value; it only performs an operation.
- **`sortByDepartment()`**: The method name, indicating that it will handle the sorting and displaying of employees based on their department.

```
if (employees.isEmpty()) {

    System.out.println("No employees to display.");

    return;

}
```

- **`if (employees.isEmpty())`**: This condition checks if the `employees` list is empty. If the list is empty, it means there are no employees stored in the system.
- **`System.out.println("No employees to display.");`**: If the list is empty, this message is printed to inform the user that there are no employees to sort.

- **`return;`**: The method returns immediately, exiting before performing any further operations, since there's nothing to sort if there are no employees.

```
System.out.println("Enter the department to sort by:");

String department = scanner.nextLine();
```

- **`System.out.println("Enter the department to sort by:");`**: This line prompts the user to input the name of the department they want to sort employees by.
- **`String department = scanner.nextLine();`**: The user's input (department name) is read using `scanner.nextLine()` and stored in the `department` variable.

```
ArrayList<Employee> sameDepartmentEmployees = new ArrayList<>();
```

- **`ArrayList<Employee> sameDepartmentEmployees = new ArrayList<>();`**: This creates a new, empty `ArrayList` called `sameDepartmentEmployees`. This list will hold all employees who belong to the department entered by the user.

```
for (Employee employee : employees) {

    if (employee.getDepartment().equalsIgnoreCase(department)) {

        sameDepartmentEmployees.add(employee);

    }

}
```

- **`for (Employee employee : employees)`**: This is a **for-each loop** that iterates through each `Employee` object in the `employees` list.
- **`if (employee.getDepartment().equalsIgnoreCase(department))`**: This condition checks if the department of the current employee (using `employee.getDepartment()`) matches the department entered by the user.
    - **`equalsIgnoreCase(department)`**: This method compares two strings (in this case, department names) while ignoring case (upper or lower). This ensures that input like "HR" and "hr" are treated the same.
- **`sameDepartmentEmployees.add(employee);`**: If the employee's department matches the user's input, that employee is added to the `sameDepartmentEmployees` list.

```
sameDepartmentEmployees.sort(Comparator.comparing(Employee::getName));
```

- **`sameDepartmentEmployees.sort()`**: This line sorts the `sameDepartmentEmployees` list based on the employee names.
- **`Comparator.comparing(Employee::getName)`**: This is a comparator that compares employees by their names. The `Employee::getName` is a **method reference** that tells the comparator to use the `getName()` method of the `Employee` class to compare the employees.
    - This means that the employees will be sorted in alphabetical order by their names.

```
if (sameDepartmentEmployees.isEmpty()) {

    System.out.println("No employees found in department: " +
department);

} else {

    System.out.println("Employees in department: " + department);

    for (Employee employee : sameDepartmentEmployees) {

        employee.displayEmployee();

    }
}
```

- **`if (sameDepartmentEmployees.isEmpty())`**: This checks if the `sameDepartmentEmployees` list is empty, meaning no employees were found in the specified department.
    - If no employees match the department name entered by the user, the `sameDepartmentEmployees` list will be empty.
- **`System.out.println("No employees found in department: " + department);`**:
    - If no employees were found, this message is printed, informing the user that there are no employees in the specified department.
- **`else`**: If the list is not empty, meaning employees were found in the specified department, the program moves on to display those employees.
- **`System.out.println("Employees in department: " + department);`**:
    - A message is printed, stating that the following employees are from the department the user specified.
- **`for (Employee employee : sameDepartmentEmployees)`**: This is a for-each loop that iterates through the `sameDepartmentEmployees` list (which now contains only the employees from the specified department).
- **`employee.displayEmployee();`**: This calls the `displayEmployee()` method for each employee in the list, which prints the employee's details (ID, name, designation, salary, and department).

How the Method Works

1. **Check if Employees Exist**: The method first checks if there are any employees in the system. If the list is empty, it exits early.
2. **Ask for Department**: The user is asked to input the department by which they want to filter employees.
3. **Filter Employees**: The method iterates through the list of employees and filters out those that belong to the specified department.
4. **Sort Employees**: The filtered list of employees is sorted by their names in alphabetical order.
5. **Display Sorted Employees**: If any employees were found in the specified department, they are displayed. If no employees were found, an appropriate message is shown.

Summary

- The `sortByDepartment()` method filters and sorts employees by a department specified by the user.

- It searches the `employees` list, collects the employees from the specified department, sorts them by name, and then displays the sorted list.
- If no employees are found in the department, it informs the user that no matches were found.

Menu to Handle Operations

```
public void showMenu() {
    int choice;


    do {
        System.out.println("\n*** Employee Management System ***");
        System.out.println("1. Add Employee");
        System.out.println("2. View Employees");
        System.out.println("3. Update Employee");
        System.out.println("4. Delete Employee");
        System.out.println("5. Sort Employees by Department");
        System.out.println("6. Exit");
        System.out.print("Enter your choice: ");
        choice = scanner.nextInt();
        scanner.nextLine(); // consume newline


        switch (choice) {
            case 1:
                addEmployee();
                break;
            case 2:
                viewEmployees();

                break;

            case 3:

                updateEmployee();

                break;

            case 4:

                deleteEmployee();

                break;

            case 5:

                sortByDepartment();

                break;

            case 6:

                System.out.println("Exiting the system.");

                break;

            default:

                System.out.println("Invalid choice! Please try again.");

        }

    } while (choice != 6);

}
```

showMenu(): This method presents a menu with options for adding, viewing, updating, deleting employees, sorting employees by department, and exiting the program. It loops until the user selects "6" to exit.

Main Class

```
public class Main {

    public static void main(String[] args) {

        EmployeeManagementSystem ems = new EmployeeManagementSystem();

        ems.showMenu();

    }

}
```

- The `Main` class contains the `main()` method, which creates an instance of `EmployeeManagementSystem` and starts the program by calling the `showMenu()` method.