Practices for Lesson 5

Practices Overview

In these practices, you will use Java interfaces and apply design patterns.

Practice 5-1: Summary Level: Implementing an Interface

Overview

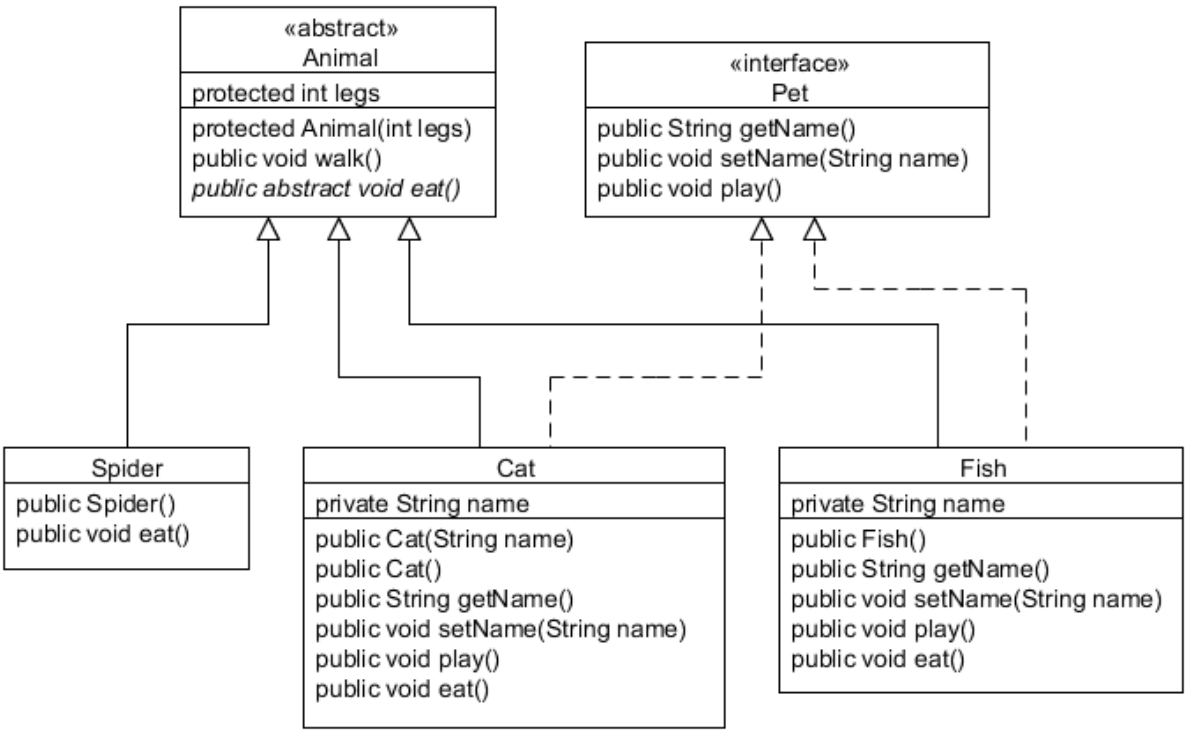
In this practice, you will create an interface and implement that interface.

Assumptions

You have reviewed the interface section of this lesson.

Summary

You have been given a project that contains an abstract class named `Animal`. You create a hierarchy of animals that is rooted in the `Animal` class. Several of the animal classes implement an interface named `Pet`, which you will create.



Tasks

1. Open the `Pet` project as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\05\practices`. (or you other directory)
   c. Select `Pet` and select the “Open as Main Project” check box.
   d. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see text displayed in the output window.

4. Review the `Animal` and `Spider` classes.
    a. Open the `Animal.java` file (under the `com.example` package).
    b. Review the abstract `Animal` class. You will extend this class.
    c. Open the `Spider.java` file (under the `com.example` package).
    d. The `Spider` class is an example of extending the `Animal` class.
5. Create a new Java interface: `Pet` in the `com.example` package.
6. Code the `Pet` interface. This interface should include three method signatures:
    - `public String getName();`
    - `public void setName(String name);`
    - `public void play();`
7. Create a new Java class: `Fish` in the `com.example` package.
8. Code the `Fish` class.
    a. This class should:
        - Extend the `Animal` class
        - Implement the `Pet` interface
    b. Complete this class by creating:
        - A String field called `name`
        - Getter and setter methods for the `name` field
        - A no-argument constructor that passes a value of 0 to the parent constructor
        - A `play()` method that prints out `"Just keep swimming."`
        - An `eat()` method that prints out `"Fish eat pond scum."`
        - A `walk()` method that overrides the `Animal` class `walk` method. It should first call the super class `walk` method, and then print `"Fish, of course, can't walk; they swim."`
9. Create a new Java class: `Cat` in the `com.example` package.
10. Code the `Cat` class.
    a. This class should:
        - Extend the `Animal` class
        - Implement the `Pet` interface
    b. Complete this class by creating:
        - A String field called `name`
        - Getter and setter methods for the `name` field
        - A constructor that receives a name String and passes a value of `4` to the parent constructor
        - A no-argument constructor that passes a value of `"Fluffy"` to the other constructor in this class
        - A `play()` method that prints out `name + " likes to play with string."`
        - An `eat()` method that prints out `"Cats like to eat spiders and fish."`

---

11. Modify the `PetMain` class.
    a. Open the `PetMain.java` file (under the `com.example` package).
    b. Review the main method. You should see the following lines of code:

```
Animal a;
//test a spider with a spider reference
Spider s = new Spider();
s.eat();
s.walk();
//test a spider with an animal reference
a = new Spider();
a.eat();
a.walk();
```

    c. Add additional lines of code to test the `Fish` and `Cat` classes that you created.
        - Try using every constructor
        - Experiment with using every reference type possible and determine which methods can be called with each type of reference. Use a `Pet` reference while testing the `Fish` and `Cat` classes.
    d. Implement and test the `playWithAnimal(Animal a)` method.
        - Determine whether the argument implements the `Pet` interface. If so, cast the reference to a `Pet` and invoke the `play` method. If not, print a message of "`Danger! Wild Animal`".
        - Call the `playWithAnimal(Animal a)` method from within `main`, passing in each type of animal.
12. Run the project. You should see text displayed in the output window.

Practice 5-1: Detailed Level: Implementing an Interface

Overview

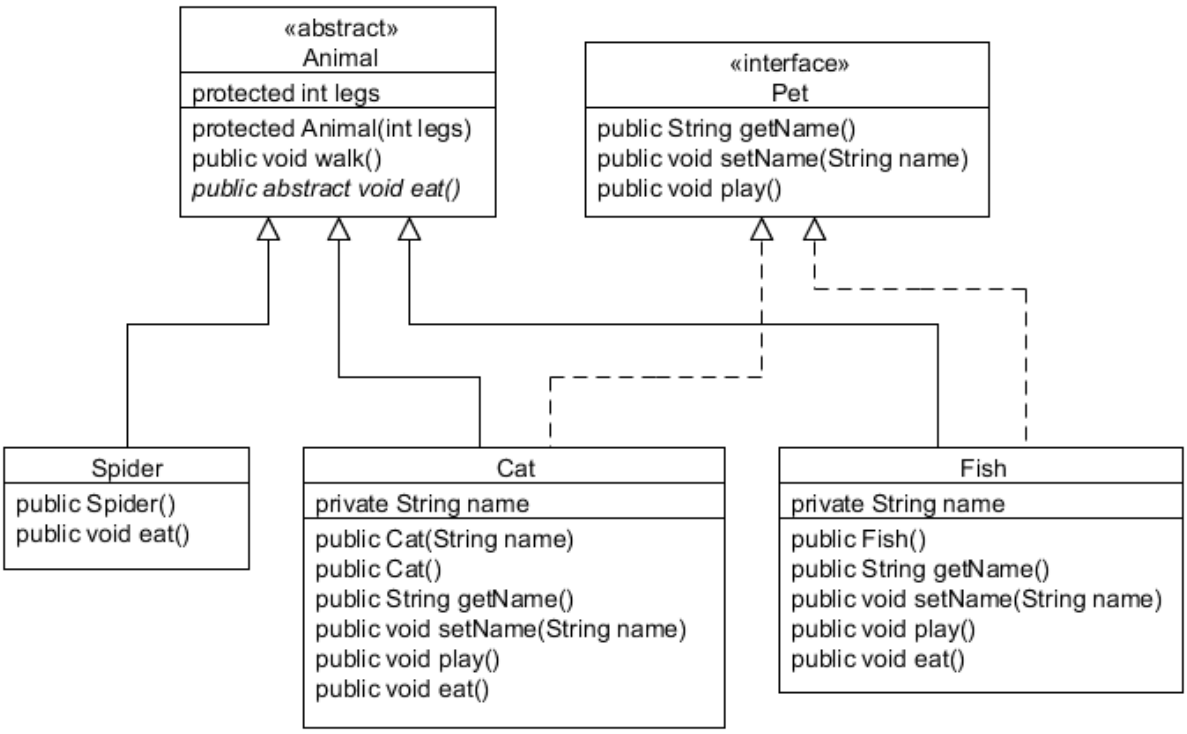
In this practice, you will create an interface and implement that interface.

Assumptions

You have reviewed the interface section of this lesson.

Summary

You have been given a project that contains an abstract class named `Animal`. You create a hierarchy of animals that is rooted in the `Animal` class. Several of the animal classes implement an interface named `Pet`, which you will create.



Tasks

1. Open the `Pet` project as the main project.
   a. Select File > Open Project.
   a. Browse to `D:\labs\05\practices`. (or you other directory)
   b. Select `Pet` and select the "Open as Main Project" check box.
   c. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see text displayed in the output window.

4. Review the `Animal` and `Spider` classes.
   a. Open the `Animal.java` file (under the `com.example` package).
   b. Review the abstract `Animal` class. You will extend this class.
   c. Open the `Spider.java` file (under the `com.example` package).
   d. The `Spider` class is an example of extending the `Animal` class.
5. Create a new Java interface: `Pet` in the `com.example` package.
6. Code the `Pet` interface. This interface should include three method signatures:

```
public String getName();
public void setName(String name);
public void play();
```

7. Create a new Java class: `Fish` in the `com.example` package.
8. Code the `Fish` class.
   a. This class should extend the `Animal` class and implement the `Pet` interface.

```
public class Fish extends Animal implements Pet
```

   b. Complete this class by creating:
      - A String field called `name`.

```
private String name;
```

      - Getter and setter methods for the `name` field.

```
@Override
public String getName() {
    return name;
}

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

      - A no-argument constructor that passes a value of 0 to the parent constructor.

```
public Fish() {
    super(0);
}
```

      - A `play()` method that prints out `"Just keep swimming."`

```
@Override
public void play() {
    System.out.println("Just keep swimming.");
}
```

- An `eat()` method that prints out `"Fish eat pond scum."`

```
@Override
public void eat() {
    System.out.println("Fish eat pond scum.");
}
```

- A `walk()` method that overrides the `Animal` class `walk` method. It should first call the super class `walk` method, and then print `" Fish, of course, can't walk; they swim."`

```
@Override
public void walk() {
    super.walk();
    System.out.println("Fish, of course, can't walk; they
swim.");
}
```

9. Create a new Java class: `Cat` in the `com.example` package.
10. Code the `Cat` class.
    a. This class should extend the `Animal` class and implement the `Pet` interface.

```
public class Cat extends Animal implements Pet
```

    b. Complete this class by creating:
    - A String field called `name`.
    - Getter and setter methods for the `name` field.
    - A constructor that receives a name String and passes a value of `4` to the parent constructor.

```
public Cat(String name) {
    super(4);
    this.name = name;
}
```

    - A no-argument constructor that passes a value of `"Fluffy"` to the other constructor in this class.

```
public Cat() {
    this("Fluffy");
}
```

    - A `play()` method that prints out `name + " likes to play with string."`

```
@Override
public void play() {
    System.out.println(name + " likes to play with string.");
}
```

    - An `eat()` method that prints out `"Cats like to eat spiders and fish."`

11. Modify the `PetMain` class.
    a. Open the `PetMain.java` file (under the `com.example` package).
    b. Review the main method. You should see the following lines of code:

```
Animal a;
//test a spider with a spider reference
Spider s = new Spider();
s.eat();
s.walk();
//test a spider with an animal reference
a = new Spider();
a.eat();
a.walk();
```

    c. Add additional lines of code to test the `Fish` and `Cat` classes that you created.
        - Try using every constructor
        - Experiment with using every reference type possible and determine which methods can be called with each type of reference. Use a `Pet` reference while testing the `Fish` and `Cat` classes.

```
Pet p;

Cat c = new Cat("Tom");
c.eat();
c.walk();
c.play();
a = new Cat();
a.eat();
a.walk();
p = new Cat();
p.setName("Mr. Whiskers");
p.play();

Fish f = new Fish();
f.setName("Guppy");
f.eat();
f.walk();
f.play();
a = new Fish();
a.eat();
a.walk();
```

    d.

Implement and test the `playWithAnimal(Animal a)` method.

- Determine whether the argument implements the `Pet` interface. If so, cast the reference to a `Pet` and invoke the `play` method. If not, print a message of "`Danger! Wild Animal`".

```
public static void playWithAnimal(Animal a) {
    if(a instanceof Pet) {
        Pet p = (Pet)a;
        p.play();
    } else {
        System.out.println("Danger! Wild Animal");
    }
}
```

- Call the `playWithAnimal(Animal a)` method at the end of the `main` method, passing in each type of animal.

```
playWithAnimal(s);
playWithAnimal(c);
playWithAnimal(f);
```

12. Run the project. You should see text displayed in the output window.

Practice 5-2: Summary Level: Applying the DAO Pattern

Overview

In this practice, you will take an existing application and refactor the code to implement the data access object (DAO) design pattern.

Assumptions

You have reviewed the DAO sections of this lesson.

Summary

You have been given a project that implements the logic for a human resources application. The application allows for creating, retrieving, updating, deleting, and listing `Employee` objects.

`Employee` objects are currently stored in-memory using an array. You must move any code related to the persistence of `Employee` objects out of the `Employee` class. In later practices, you will supply alternative persistence implementations. In the future, this application should require no modification when substituting the persistence implementation.

Tasks

1. Open the `EmployeeMemoryDAO` project as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\05\practices`. (or you other directory)
   b. Select `EmployeeMemoryDAO` and select the "Open as Main Project" check box.
   c. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see a menu. Test all the menu choices.

   ```
   [C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit:
   ```

   Note: When entering dates, they should be in the form of: Nov 26, 1976

   Employee IDs should be in the range of 0 through 9.
4. Review the `Employee` class.
   a. Open the `Employee.java` file (under the `com.example.model` package).
   b. Find the array used to store employees.

   ```
   private static Employee[] employeeArray = new Employee[10];
   ```

   Note: The employee's `id` is used as the array index.
   c. Locate any methods that utilize the `employeeArray` field. These methods are used to persist employee objects.
5. Create a new `com.example.dao` package.
6. Create an `EmployeeDAO` interface in the `com.example.dao` package.

7. Complete the `EmployeeDAO` interface with the following method signatures.

```
public void add(Employee emp);
public void update(Employee emp);
public void delete(int id);
public Employee findById(int id);
public Employee[] getAllEmployees();
```

8. Create an `EmployeeDAOMemoryImpl` class in the `com.example.dao` package.
9. Complete the `EmployeeDAOMemoryImpl` class.
   a. Move the `employeeArray` and any related methods from the `Employee` class to the `EmployeeDAOMemoryImpl` class.
   b. Implement the `EmployeeDAO` interface. Modify the methods that you moved in the previous step to become the methods required by the `EmployeeDAO` interface.

      Hint: In this DAO, the `add` and `update` methods will function the same.
10. Update the `EmployeeTestInteractive` class.
    a. The `EmployeeTestInteractive` class no longer compiles, review the errors.
    b. Create an `EmployeeDAO` instance in the `main` method. Use the `EmployeeDAO` interface as the reference type.
    c. Modify any lines containing errors to use the `EmployeeDAO` instance.
11. Run the project. You should see a menu. Test all the menu choices.

    Note: While functional, the `EmployeeTestInteractive` class is still tied to a specific type of DAO because it references the `EmployeeDAO` implementing class by name.

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

    In the following steps, you remove this tight coupling from the `EmployeeTestInteractive` class by creating a DAO factory.
12. Modify the `EmployeDAOMemoryImpl` interface.
    - Add a protected, no-argument constructor.
13. Create an `EmployeeDAOFactory` class in the `com.example.dao` package.
14. Complete the `EmployeeDAOFactory` class.
    - Add a method that returns an `EmployeeDAO`.

```
public EmployeeDAO createEmployeeDAO() {
     return new EmployeeDAOMemoryImpl();
}
```

15. Update the `EmployeeTestInteractive` class to use the `EmployeeDAOFactory` class.
    a. Obtain an `EmployeeDAOFactory` instance in the `main` method.
    b. Obtain an `EmployeeDAO` instance using the factory created in the previous step.
16. Run the project. You should see a menu. Test all the menu choices.

    In the future, you will be able to change the persistence mechanism to use a database without changing the reference types or method calls used in the `EmployeeTestInteractive` class.

Practice 5-2: Detailed Level: Applying the DAO Pattern

Overview

In this practice, you will take an existing application and refactor the code to implement the data access object (DAO) design pattern.

Assumptions

You have reviewed the DAO sections of this lesson.

Summary

You have been given a project that implements the logic for a human resources application. The application allows for creating, retrieving, updating, deleting, and listing `Employee` objects.

`Employee` objects are currently stored in-memory using an array. You must move any code related to the persistence of `Employee` objects out of the `Employee` class. In later practices, you will supply alternative persistence implementations. In the future, this application should require no modification when substituting the persistence implementation.

Tasks

1. Open the `EmployeeMemoryDAO` project as the main project.
   a. Select File > Open Project.
   c. Browse to `D:\labs\05\practices`. (or you other directory)
   b. Select `EmployeeMemoryDAO` and select the "Open as Main Project" check box.
   c. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see a menu. Test all the menu choices.

   ```
   [C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit:
   ```

   Note: When entering dates, they should be in the form of: Nov 26, 1976

   Employee IDs should be in the range of 0 through 9.
4. Review the `Employee` class.
   a. Open the `Employee.java` file (under the `com.example.model` package).
   b. Find the array used to store employees. You will relocate this field in a subsequent step.

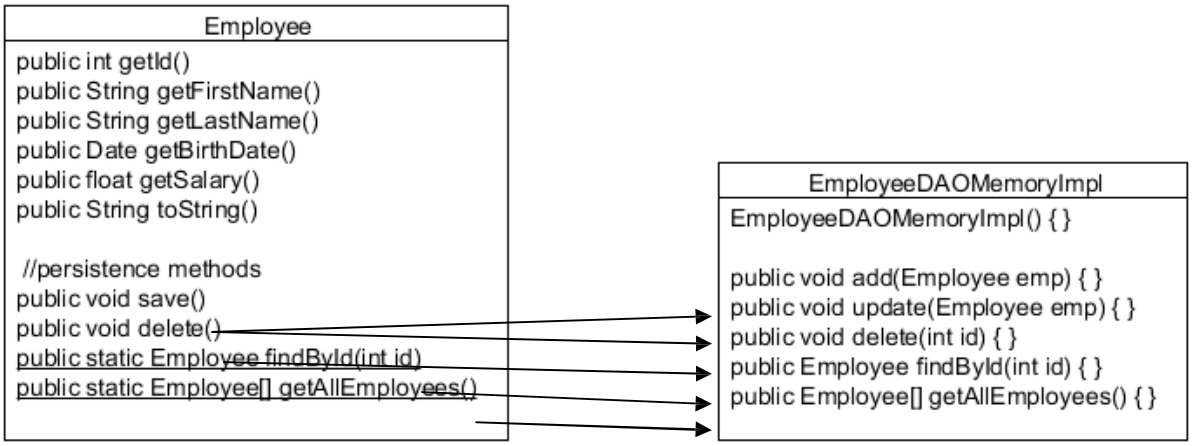
   ```
   private static Employee[] employeeArray = new Employee[10];
   ```

   Note: The employee's `id` is used as the array index.
   c. Locate the `save`, `delete`, `findById`, and `getAllEmployees` methods that utilize the `employeeArray` field. These methods are used to persist employee objects. You will relocate these methods in a subsequent step.
5. Create a new `com.example.dao` package.
6. Create an `EmployeeDAO` interface in the `com.example.dao` package.

7. Complete the `EmployeeDAO` interface with the following method signatures. Add import statements as needed.

```
public void add(Employee emp);
public void update(Employee emp);
public void delete(int id);
public Employee findById(int id);
public Employee[] getAllEmployees();
```

8. Create an `EmployeeDAOMemoryImpl` class in the `com.example.dao` package.
9. Complete the `EmployeeDAOMemoryImpl` class.
   a. Move the `employeeArray` and any related methods from the `Employee` class to the `EmployeeDAOMemoryImpl` class:



   b. Implement the `EmployeeDAO` interface. Modify the methods that you moved in the previous step to become the methods required by the `EmployeeDAO` interface.
      - The `save` method becomes the `add` method and is modified to have an `Employee` parameter.
      - The `save` method is duplicated to become the `update` method and is modified to have an `Employee` parameter.
      - The `delete` method is modified to have an `id` parameter.
      - The `findById` method is no longer `static`.
      - The `getAllEmployees` method is no longer `static`.

```
public class EmployeeDAOMemoryImpl implements EmployeeDAO {
    private static Employee[] employeeArray = new Employee[10];

    public void add(Employee emp) {
        employeeArray[emp.getId()] = emp;
    }

    public void update(Employee emp) {
        employeeArray[emp.getId()] = emp;
    }

    public void delete(int id) {
        employeeArray[id] = null;
    }

    public Employee findById(int id) {
        return employeeArray[id];
    }

    public Employee[] getAllEmployees() {
        List<Employee> emps = new ArrayList<>();
        for (Employee e : employeeArray) {
            if (e != null) {
                emps.add(e);
            }
        }
        return emps.toArray(new Employee[0]);
    }
}
```

10. Update the `EmployeeTestInteractive` class.
    a. The `EmployeeTestInteractive` class no longer compiles, review the errors.
    b. Create an `EmployeeDAO` instance in the `main` method. Use the `EmployeeDAO` interface as the reference type. Replace the line:

```
//TODO create dao
```

With:

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

    c.

Modify any lines containing errors to use the `EmployeeDAO` instance. For example:

```
case 'C':
    emp = inputEmployee(in);
    dao.add(emp);
    System.out.println("Successfully added Employee Record: " +
emp.getId());
    System.out.println("\n\nCreated " + emp);
    break;
```

Note: You can also remove the now unnecessary finding of employee object that is present when deleting an employee.

```
// Find this Employee record
emp = null;
emp = Employee.findById(id);
if (emp == null) {
    System.out.println("\n\nEmployee " + id + " not found");
    break;
}
```

11. Run the project. You should see a menu. Test all the menu choices.

    Note: While functional, the `EmployeeTestInteractive` class is still tied to a specific type of DAO because it references the `EmployeeDAO` implementing class by name.

    ```
    EmployeeDAO dao = new EmployeeDAOMemoryImpl();
    ```

    In the following steps, you remove this tight coupling from the `EmployeeTestInteractive` class by creating a DAO factory.

12. Modify the `EmployeDAOMemoryImpl` interface.

    a. Add a protected, no-argument constructor.

    ```
    EmployeeDAOMemoryImpl() {
    }
    ```

13. Create an `EmployeeDAOFactory` class in the `com.example.dao` package.

14. Complete the `EmployeeDAOFactory` class.

    - Add a method that returns an `EmployeeDAO`.

    ```
    public class EmployeeDAOFactory {

        public EmployeeDAO createEmployeeDAO() {
            return new EmployeeDAOMemoryImpl();
        }
    }
    ```

15. Update the `EmployeeTestInteractive` class to use the `EmployeeDAOFactory` class.

    a. Obtain an `EmployeeDAOFactory` instance in the `main` method. Replace the line:

    ```
    //TODO create factory
    ```

    With:

    ```
    EmployeeDAOFactory factory = new EmployeeDAOFactory();
    ```

    b. Obtain an `EmployeeDAO` instance using the factory created in the previous step. Replace the line:

    ```
    EmployeeDAO dao = new EmployeeDAOMemoryImpl();
    ```

    With:

    ```
    EmployeeDAO dao = factory.createEmployeeDAO();
    ```

    c. Fix any imports.

16. Run the project. You should see a menu. Test all the menu choices.

    In the future, you will be able to change the persistence mechanism to use a database without changing the reference types or method calls used in the `EmployeeTestInteractive` class. Notice that none of the *`MemoryImpl` classes are used by name from within the `EmployeeTestInteractive` class.

(Optional) Practice 5-3: Implementing Composition

Overview

In this practice, you will take an existing application and refactor it to make use of composition.

Assumptions

You have reviewed the interface and composition sections of this lesson.

Summary

You have been given a small project that represents a hierarchy of animals that is rooted in the `Animal` class. Several of the animal classes implement an interface named `Pet`. This project is a completed implementation of the “Implementing an Interface” practice.

There are some potential problems with the design of the existing project. If you wanted to restrict a pet’s name to less than 20 characters how many classes would you have to modify? Would this problem become worse with the addition of new animals?

If some types of animals, such as Fish, cannot walk, should they have a walk method?

Tasks

1. Open the `PetComposition` project as the main project.
   a. Select File > Open Project.
   d. Browse to `D:\labs\05\practices`. (or you other directory)
   b. Select `PetComposition` and select the "Open as Main Project" check box.
   c. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see output in the output window.
4. Centralize all “name” functionality.

   All pets can be named, but you may want to give names to objects that cannot play. For instance, you could name a volleyball “Wilson.” Your design should reflect this.

   a. Create a `Nameable` interface (under the `com.example` package).
   b. Complete the `Nameable` interface with `setName` and `getName` method signatures.

```
public interface Nameable {

    public void setName(String name);

    public String getName();

}
```

   c. Create a `NameableImpl` class (under the `com.example` package).
   d.

Complete the `NameableImpl` class. It should:

- Implement the `Nameable` interface
- Contain a private String field called `name`
- Only accept names less than 20 characters in length
- Print "`Name too long`" if a name is too long

e. Modify the `Pet` interface.

- Extend the `Nameable` interface.
- Remove the `getName` and `setName` method signatures (they are inherited now).

f. Modify the `Fish` and `Cat` classes to use composition.

- Delete the `name` field.
- Delete the existing `getName` and `setName` methods.
- Add a new `Nameable` field.

```
private Nameable nameable = new NameableImpl();
```

- Add `getName` and `setName` methods that delegate to the `Nameable` field.

  Hint: Position the cursor within the curly braces for the class. Open the Source menu, select Insert Code, select Delegate Method, select the Nameable check box, and click the Generate button.

- Replace any use of the old `name` field with calls to the `getName` and `setName` methods.

5. Centralize all walking functionality.

   Only some animal can walk. Remove the `walk` method from the `Animal` class and use interfaces and composition to facilitate walking.

   a. Create an `Ambulatory` interface (under the `com.example` package).

   b. Complete the `Ambulatory` interface with the `walk` method signature.

```
public interface Ambulatory {

    public void walk();

}
```

   c. Create an `AmbulatoryImpl` class (under the `com.example` package).

   d. Complete the `AmbulatoryImpl` class. It should:

- Implement the `Ambulatory` interface
- Contain a private int field called `legs`
- Contain a single argument constructor that receives an `int` value to be stored in the `legs` field
- Contain a walk method

```
public void walk() {
    System.out.println("This animal walks on " + legs + "
legs.");
}
```

---

e. Delete the `walk` method from the `Fish` class.

f. Modify the `Spider` and `Cat` classes to use composition.

- Add a new `Ambulatory` field.

```
private Ambulatory ambulatory;
```

- Add a `walk` method that delegates to the `Ambulatory` field.

  Hint: Position the cursor within the curly braces for the class. Open the Source menu, select Insert Code, select Delegate Method, select the Ambulatory check box, and click the Generate button.

g. Initialize the ambulatory field within the `Spider` and `Cat` constructors. For example:

```
public Spider() {
    ambulatory = new AmbulatoryImpl(8);
}
```

6. Modify the `PetMain` class to test the `walk` method. The `walk` method can only be called on `Spider`, `Cat`, or `Ambulatory` references.