Practices for Lesson 6: Interfaces and Lambda Expressions

Chapter 6

Practices for Lesson 6: Overview

Practices Overview

In these practices, you will use Java interfaces and lambda expressions.

Practice 6-1: Summary Level: Implementing an Interface

Overview

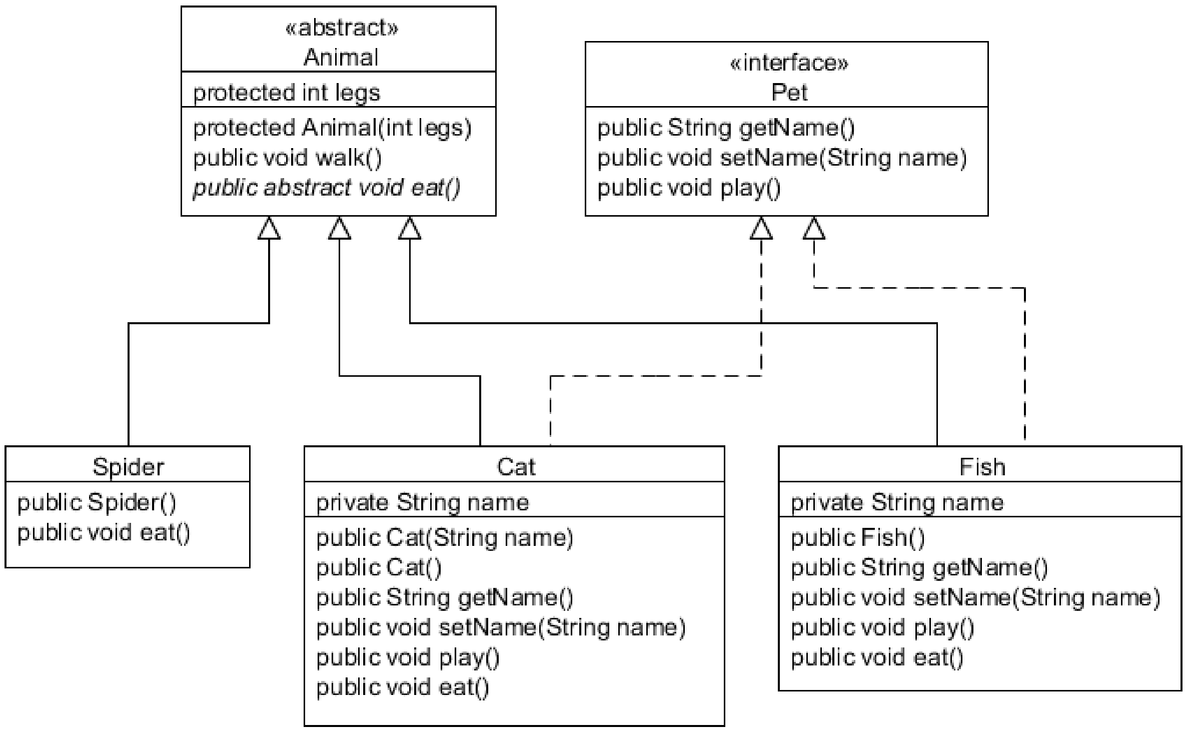
In this practice, you will create an interface and implement that interface.

Assumptions

You have reviewed the interface section of this lesson.

Summary

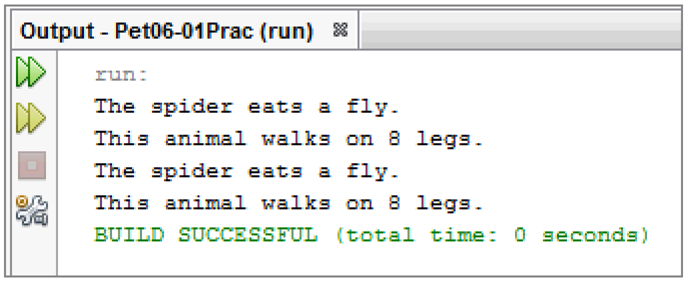
You have been given a project that contains an abstract class named `Animal`. You create a hierarchy of animals that is rooted in the `Animal` class. Several of the animal classes implement an interface named `Pet`, which you will create.



Tasks

1. Open the `Pet06-01Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-Interfaces/practices/practice1`.
   c. Select `Pet06-01Prac` click Open Project.
2. Expand the project directories.

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



```
run:
The spider eats a fly.
This animal walks on 8 legs.
The spider eats a fly.
This animal walks on 8 legs.
BUILD SUCCESSFUL (total time: 0 seconds)
```

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."`

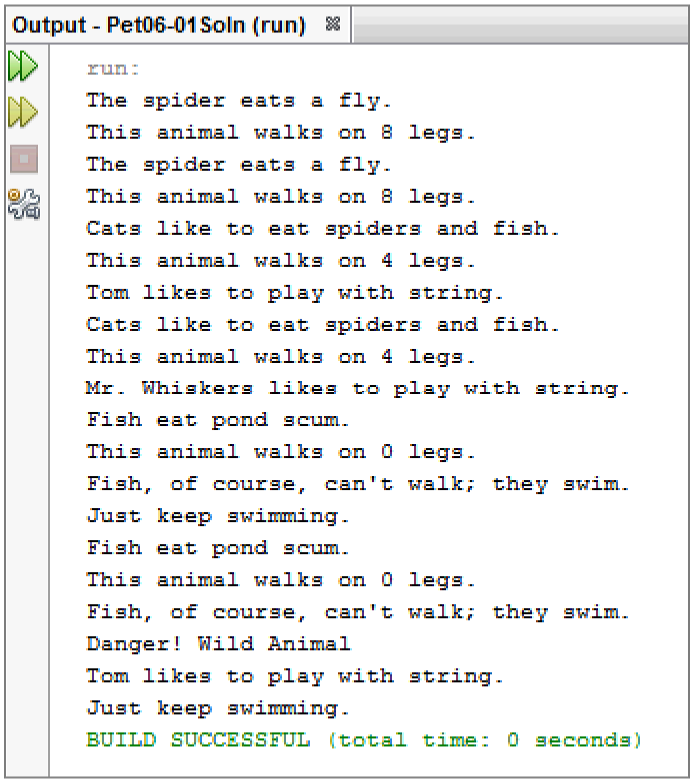
11. Modify the `PetMain` class.
    a. Open the `PetMain.java` file (under the `com.example` package).
    b. Review the main method. You should see the following lines of code:

```
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.



```
run:
The spider eats a fly.
This animal walks on 8 legs.
The spider eats a fly.
This animal walks on 8 legs.
Cats like to eat spiders and fish.
This animal walks on 4 legs.
Tom likes to play with string.
Cats like to eat spiders and fish.
This animal walks on 4 legs.
Mr. Whiskers likes to play with string.
Fish eat pond scum.
This animal walks on 0 legs.
Fish, of course, can't walk; they swim.
Just keep swimming.
Fish eat pond scum.
This animal walks on 0 legs.
Fish, of course, can't walk; they swim.
Danger! Wild Animal
Tom likes to play with string.
Just keep swimming.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practice 6-1: Detailed Level: Implementing an Interface

Overview

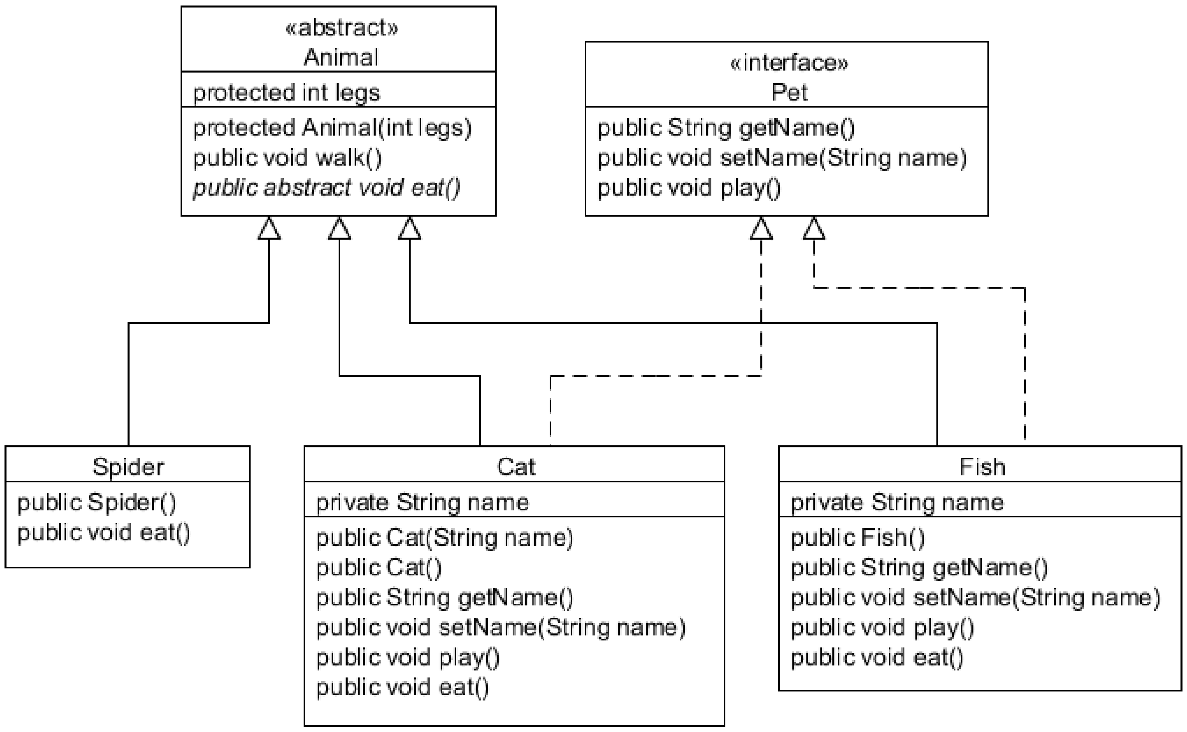
In this practice, you will create an interface and implement that interface.

Assumptions

You have reviewed the interface section of this lesson.

Summary

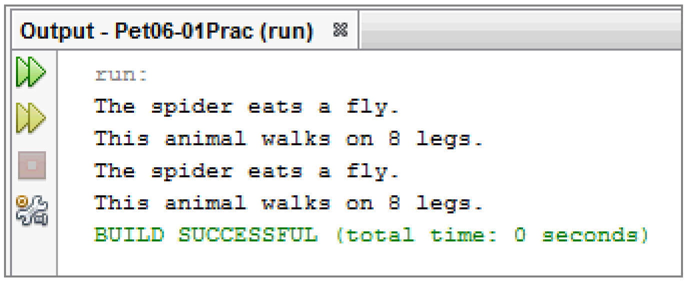
You have been given a project that contains an abstract class named `Animal`. You create a hierarchy of animals that is rooted in the `Animal` class. Several of the animal classes implement an interface named `Pet`, which you will create.



Tasks

1. Open the `Pet06-01Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-Interfaces/practices/practice1`.
   c. Select `Pet06-01Prac` and click Open Project.
2. Expand the project directories.

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



```
run:
The spider eats a fly.
This animal walks on 8 legs.
The spider eats a fly.
This animal walks on 8 legs.
BUILD SUCCESSFUL (total time: 0 seconds)
```

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.

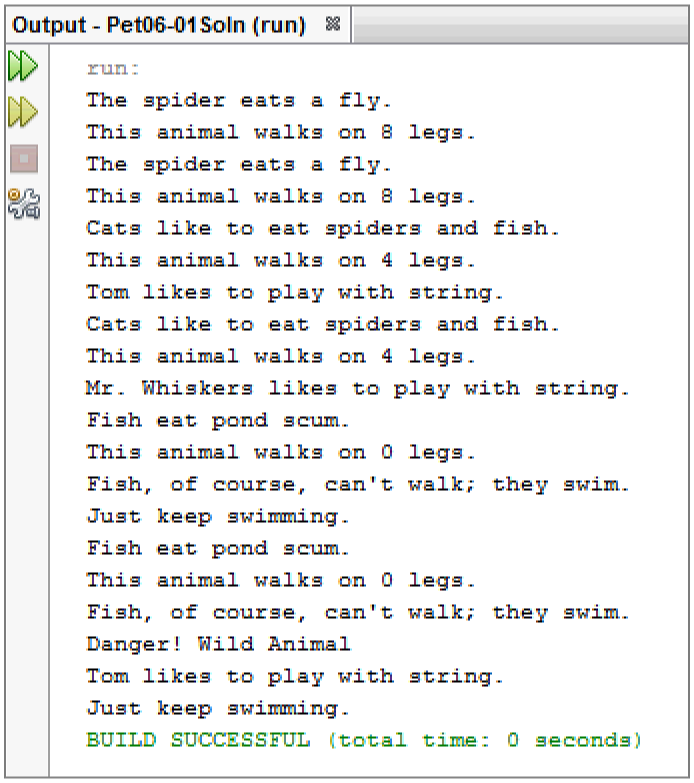
- Determine whether the argument implements the `Pet` interface. If so, cast the reference to a `Pet` and invoke the `play` method. If not, print a message of `"Danger! Wild Animal"`.

```
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.


Output - Pet06-01Soln (run)

```
run:
The spider eats a fly.
This animal walks on 8 legs.
The spider eats a fly.
This animal walks on 8 legs.
Cats like to eat spiders and fish.
This animal walks on 4 legs.
Tom likes to play with string.
Cats like to eat spiders and fish.
This animal walks on 4 legs.
Mr. Whiskers likes to play with string.
Fish eat pond scum.
This animal walks on 0 legs.
Fish, of course, can't walk; they swim.
Just keep swimming.
Fish eat pond scum.
This animal walks on 0 legs.
Fish, of course, can't walk; they swim.
Danger! Wild Animal
Tom likes to play with string.
Just keep swimming.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practice 6-2: Summary Level: Using Java Interfaces

Overview

In this practice, you will take the existing banking application and refactor the code to use interfaces.

Assumptions

You have reviewed the `interface` section of this lesson.

Summary

You have been given a project that implements the logic for a bank. Update the application to use Java interfaces.

Tasks

1. Open the `InterfaceBanking06-02Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-interfaces/practices/practice2`.
   c. Select `InterfaceBanking06-02Prac` and click Open Project.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Two interface files have been created for you `AccountOperations.java` and `BankOperations.java`. You will update these files.

   Note: Certain steps that follow may generate a number of errors in your source files. Do not panic! The errors will be fixed as you proceed through the changes.
5. Open the `Account.java` file and the `AccountOperations.java` file.
6. Copy the following method signatures from the `Account.java` file to the `AccountOperations.java` file. Here are the method names you should copy: `getBalance()`, `deposit()`, `withdraw()`, and `getDescription()`.
7. Update `CheckingAccount.java` to use implement `AccountOperations.`
8. Update `SavingsAccount.java` to use implement `AccountOperations.`
9. In `Account.java` remove the following methods: `getBalance()`, `deposit()`, `withdraw()`, and `getDescription()`.
10. In `Account.java` update the `toString()` method to print a message without calling `getDescription()`.
11. Save `Account.java`. Close the file.
12. Edit `CheckingAccount.java`.
13. Implement a `getBalance()` method.
14. Implement a `deposit()` method.
15. Override the `toString` method.
16. Save the file. Close the file.
17. Edit `SavingsAccount.java`.
18. Implement a `getBalance()` method.
19. Override the `toString` method.

20. Save the file. Close the file.
21. Edit the `Bank.java` file.
22. Update `Bank.java` so that it implements the `BankOperations` class.
23. Save the file.
24. Edit the `BankOperations.java` file.
25. Copy the following method signatures from the `Bank.java` file to the `BankOperations.java` file. The methods signatures to copy are: `addCustomer()`, `getNumOfCustomers()`, and `getCustomer()`.
26. Save the file.
27. Open the `CustomerReport.java` file.
28. Copy the `generateReport()` method to the `BankOperations.java` file.
29. In the newly copied method, change any reference to `bank` to `this`.
30. Save the `BankOperations.java` file.
31. Delete the `CustomerReport.java` file.
32. Open the `Main.java` file.
33. Change the type definition of `bank` to the new interface `BankOperations`.
34. Change the code to call the `generateReport` method from `bank`.
35. Run the project. Everything should print again.
36. Edit the `Customer.java` file.
37. Change the `Account[]` array to an `AccountOperations[]` array.
38. Fix any resulting errors by changing the references from `Account` to `AccountOperations`.
39. Save the file.
40. Fix the reference error in `BankOperations` caused by this change. Save the file.
41. Edit the `Main.java`.
42. Change any Checking or Savings account references to `AccountOperations` references.
    Hint: Changes should be made to accounts: 1 and 5

43. Run the project. The output should look like the following:

```
              CUSTOMERS REPORT
              ================

Customer: Smith, Will
Branch: LA, Basic
    Savings Account balance is 500.0

Customer: Cooper, Bradley
Branch: Boston, Loan
    Savings Account balance is 1060.0

Customer: Simms, Jane
Branch: Mumbai, Full
    Checking Account balance is 200.0

Customer: Bryant, Owen
Branch: Bangalore, Full
    Checking Account balance is 200.0

Customer: Soley, Tim
Branch: LA, Basic
    Checking Account balance is 200.0

Customer: Soley, Maria
Branch: Bangalore, Full
    Checking Account balance is 100.0
```

Practice 6-2: Detailed Level: Using Java Interfaces

Overview

In this practice, you will take an existing application and refactor the code to use interfaces.

Assumptions

You have reviewed the `interface` section of this lesson.

Summary

You have been given a project that implements the logic for a bank. Update the application to use Java interfaces.

Tasks

1. Open the `InterfaceBanking06-02Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-interfaces/practices/practice2`.
   c. Select `InterfaceBanking06-02Prac` and click Open Project.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Two interface files have been created for you `AccountOperations.java` and `BankOperations.java`. You will update these files.

   Note: Certain steps that follow may generate a number of errors in your source files. Do not panic! The errors will be fixed as you proceed through the changes.
5. Open the `Account.java` file and the `AccountOperations.java` file.
6. Copy the following method signatures from the `Account.java` file to the `AccountOperations.java` file. Here are the method names you should copy: `getBalance()`, `deposit()`, `withdraw()`, and `getDescription()`.
7. Update `CheckingAccount.java` to use implement `AccountOperations`.

```
public class CheckingAccount extends Account implements
AccountOperations
```

8. Update `SavingsAccount.java` to use implement `AccountOperations`.

```
public class SavingsAccount extends Account implements
AccountOperations
```

9. In `Account.java` remove the following methods: `getBalance()`, `deposit()`, `withdraw()`, and `getDescription()`.
10. In `Account.java` update the `toString()` method to print a message without calling `getDescription()`.

```
return "Current balance is "  + balance;
```

11. Save `Account.java`. Close the file.
12. Edit `CheckingAccount.java`.

13. Implement a `getBalance()` method.

```
    @Override
    public double getBalance(){
      return balance;
    }
```

14. Implement a `deposit()` method.

```
    @Override
    public void deposit(double amount) {
       balance += amount;
    }
```

15. Override the `toString` method.

```
    @Override
    public String toString() {
      return this.getDescription() +" balance is " + balance;
    }
```

16. Save the file. Close the file.
17. Edit `SavingsAccount.java`.
18. Implement a `getBalance()` method.

```
    @Override
    public double getBalance(){
      return balance;
    }
```

19. Override the `toString` method.

```
    @Override
    public String toString() {
      return this.getDescription() +" balance is " + balance;
    }
```

20. Save the file. Close the file.
21. Edit the `Bank.java` file.
22. Update `Bank.java` so that it implements the `BankOperations` class.

```
public class Bank implements BankOperations
```

23. Save the file.
24. Edit the `BankOperations.java` file.
25. Copy the following method signatures from the `Bank.java` file to the `BankOperations.java` file. The methods signatures to copy are: `addCustomer()`, `getNumOfCustomers()`, and `getCustomer()`.
26. Save the file.
27. Open the `CustomerReport.java` file.
28. Copy the `generateReport()` method to the `BankOperations.java` file.
    - Change the method signature in `BankOperations.java` to:

```
public default void generateReport()
```

29. In the newly copied method, change any `bank` references to `this`.
30. Save the `BankOperations.java` file.

31. Delete the `CustomerReport.java` file.
32. Open the `Main.java` file.
33. Change the definition of `bank` to the following:

```
BankOperations bank = new Bank();
```

34. Change the code to call the `generateReport` method from bank.
    - Replace these lines:

```
CustomerReport report = new CustomerReport();
report.setBank(bank);
report.generateReport();
```

    - with this line:

```
bank.generateReport();
```

35. In the same file, update the `initializeCustomers(BankOperations bank)` method. Make the method `static` and note that a `BankOperations` object is passed in.
36. Save the file.
37. Run the project. Everything should print again.
38. Edit the `Customer.java` file.
39. Change the `Account[]` array to an `AccountOperations[]` array.
40. Fix any resulting errors by changing the references from `Account` to `AccountOperations`.
41. Save the file.
42. Fix the reference error in `BankOperations` caused by this change. Save the file.
43. Edit the `Main.java`.
44. Change any Checking or Savings account references to `AccountOperations` references.
    Hint: Changes should be made to accounts: 1 and 5

45. Run the project. The output should look like the following:

```
              CUSTOMERS REPORT
              ================

Customer: Smith, Will
Branch: LA, Basic
    Savings Account balance is 500.0

Customer: Cooper, Bradley
Branch: Boston, Loan
    Savings Account balance is 1060.0

Customer: Simms, Jane
Branch: Mumbai, Full
    Checking Account balance is 200.0

Customer: Bryant, Owen
Branch: Bangalore, Full
    Checking Account balance is 200.0

Customer: Soley, Tim
Branch: LA, Basic
    Checking Account balance is 200.0

Customer: Soley, Maria
Branch: Bangalore, Full
    Checking Account balance is 100.0
```

Practice 6-3: Summary Level: Write Lambda Expressions

Overview

In this practice, write additional lambda expressions for the `StringAnalzyer` application.

Assumptions

You have reviewed the lambda expressions section of this lesson.

Summary

Use the `StringAnalyzer` project from the lecture to create 3 additional lambda expressions.

Tasks

1. Open the `LambdaBasics06-03Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-interfaces/practices/practice3`.
   c. Select `LambdaBasics06-03Prac` and click Open Project.
2. Expand the project directories.
3. Open the `LambdaTest.java` file.
4. Write a lambda expression that displays strings that end with the search string.
5. Write a lambda expression that displays strings that contain the search string and are 5 characters or less in length.
6. Write a lambda expression that displays strings that contain the search string and are greater than 5 characters in length.
7. Run the project. The output should be as follows:

```
Searching for: to
==Contains==
Match: tomorrow
Match: toto
Match: to
Match: timbukto
==Starts With==
Match: tomorrow
Match: toto
Match: to
==Equals==
Match: to
==Ends With==
Match: toto
Match: to
Match: timbukto
==Less than 5==
Match: toto
Match: to
==Greater than 5==
Match: tomorrow
Match: timbukto
```

Practice 6-3: Detailed Level: Write Lambda Expressions

Overview

In this practice, write additional lambda expressions for the `StringAnalzyer` application.

Assumptions

You have reviewed the lambda expressions section of this lesson.

Summary

Use the `StringAnalyzer` project from the lecture to create three additional lambda expressions.

Tasks

1. Open the `LambdaBasics06-03Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-interfaces/practices/practice3`.
   c. Select `LambdaBasics06-03Prac` and click Open Project.
2. Expand the project directories.
3. Open the `LambdaTest.java` file
4. Write a lambda expression that displays strings that end with the search string.

```
        (t,s) -> t.endsWith(s));
```

5. Write a lambda expression that displays strings that contain the search string and are 5 characters or less in length.

```
        (t,s) -> t.contains(s) && t.length() < 5);
```

6. Write a lambda expression that displays strings that contain the search string and are greater than five characters in length.

```
        (t,s) -> t.contains(s) && t.length() > 5);
```

7. Run the project. The output should be as follows:

```
Searching for: to
==Contains==
Match: tomorrow
Match: toto
Match: to
Match: timbukto
==Starts With==
Match: tomorrow
Match: toto
Match: to
==Equals==
Match: to
==Ends With==
Match: toto
Match: to
Match: timbukto
==Less than 5==
Match: toto
Match: to
==Greater than 5==
Match: tomorrow
Match: timbukto
```