Practices for Lesson 5: Abstract and Nested Classes

Chapter 5

Practices for Lesson 5: Overview

Practices Overview

In these practices, you will use the abstract, final, and static Java keywords. You will also learn to use inner class as a helper class to a top level class.

Practice 5-1: Summary Level: Applying the Abstract Keyword

Overview

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

Assumptions

You have reviewed the abstract class section of this lesson.

Summary

You have been given a project that implements the logic for a bank. The banking software supports only the creation of saving accounts. You will enhance the software to support checking accounts.

Additional types of accounts might be added in the future.

Tasks

1. Open the `AbstractBanking05-01Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice1`.
   c. Select `AbstractBanking05-01Prac` and click the Open Project button.
2. Expand the project directories.
3. Review the `SavingsAccount` class.
   a. Open the `SavingsAccount.java` file (under the `com.example` package).
   b. Examine the fields and method implementations of `SavingsAccount..`
4. Review the `Account.java`, under the `com.example` package, this class is an `abstract` class. This class contains two abstract methods:

```
public abstract boolean withdraw(double amount);

public abstract String getDescription();
```

5. Create a new Java class, `CheckingAccount`, in the `com.example` package.
   a. `CheckingAccount` should be a subclass of `Account`.
   b. Add an `overDraftLimit` field to the `CheckingAccount` class.

```
private final double overDraftLimit;
```

   c. Add a `CheckingAccount` constructor that has two parameters.
      - `double balance`: Pass this value to the parent class constructor.
      - `double overDraftLimit`: Store this value in the overDraftLimit field.
   d. Add a `CheckingAccount` constructor that has one parameter. This constructor should set the `overDraftLimit` field to zero.
      - `double balance`: Pass this value to the parent class constructor.

e. Override the abstract `getDescription` method inherited from the `Account` class.

```
@Override
public String getDescription() {
    return "Checking Account";
}
```

Note: It is a good practice to add `@Override` to any method that would be overriding a parent class method.

f. Override the abstract `withdraw` method inherited from the `Account` class.

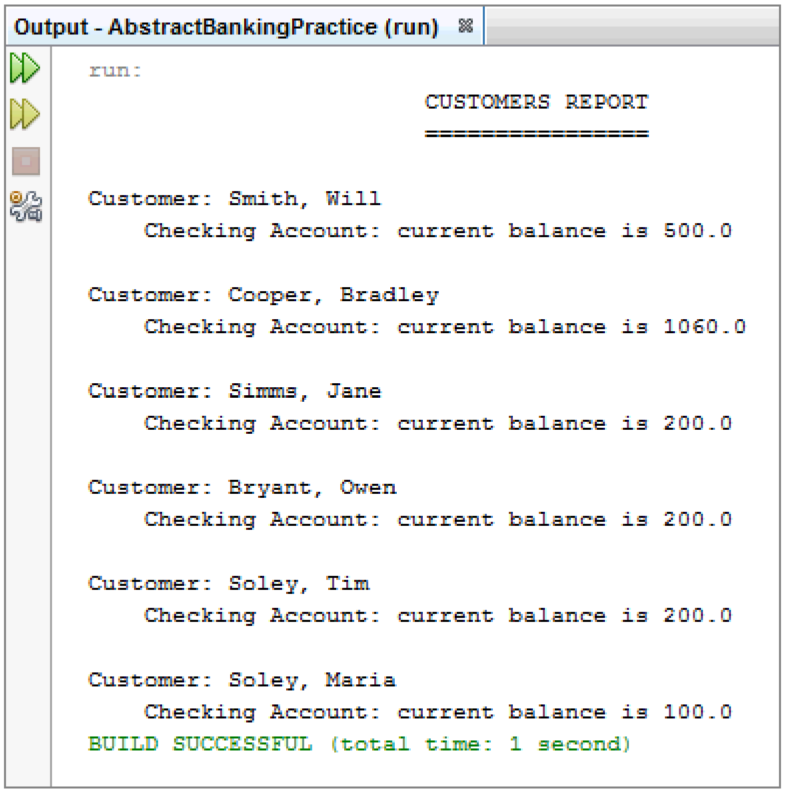
- The `withdraw` method should allow an account balance to go negative up to the amount specified in the `overDraftLimit` field.
- The `withdraw` method should return `false` if the withdraw cannot be performed, and `true` if it can.

6. Modify the `AbstractBankingMain` class to create checking accounts for the customers.

```
// Create several customers and their accounts
        bank.addCustomer("Will", "Smith");
        customer = bank.getCustomer(0);
        customer.addAccount(new SavingsAccount(500.00));

        bank.addCustomer("Bradley", "Cooper");
        customer = bank.getCustomer(1);
        SavingsAccount sack = new SavingsAccount(500.00);
        customer.addAccount(sack);
        sack.deposit(500);

        bank.addCustomer("Jane", "Simms");
        customer = bank.getCustomer(2);
        customer.addAccount(new CheckingAccount(200.00,
400.00));

        bank.addCustomer("Owen", "Bryant");
        customer = bank.getCustomer(3);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Tim", "Soley");
        customer = bank.getCustomer(4);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Maria", "Soley");
        customer = bank.getCustomer(5);
        CheckingAccount chkAcct = new CheckingAccount(100.00);
        customer.addAccount(chkAcct);
        if (chkAcct.withdraw(900.00)) {
            customer.addAccount(chkAcct);
```

```
                System.out.print(" withdraw is successful" +
chkAcct.getBalance());
           }
```

7. Run the project. You should see a report of all customers and their accounts.



```
run:
                        CUSTOMERS REPORT
                        ================

Customer: Smith, Will
    Checking Account: current balance is 500.0

Customer: Cooper, Bradley
    Checking Account: current balance is 1060.0

Customer: Simms, Jane
    Checking Account: current balance is 200.0

Customer: Bryant, Owen
    Checking Account: current balance is 200.0

Customer: Soley, Tim
    Checking Account: current balance is 200.0

Customer: Soley, Maria
    Checking Account: current balance is 100.0
BUILD SUCCESSFUL (total time: 1 second)
```

Practice 5-1: Detailed Level: Applying the Abstract Keyword

Overview

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

Assumptions

You have reviewed the abstract class section of this lesson.

Summary

You have been given a project that implements the logic for a bank. The banking software supports only the creation of saving accounts. You will enhance the software to support checking accounts. Additional types of accounts might be added in the future.

Tasks

1. Open the `AbstractBanking05-01Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice1`.
   c. Select `AbstractBanking05-01Prac`.
   d. Click Open Project.
2. Expand the project directories.
3. Review the `SavingsAccount` class.
   a. Open the `SavingsAccount.java` file (under the `com.example` package).
   b. Examine the fields and method implementations of `SavingsAccount`.
4. Review the `Account.java`, under the `com.example` package, this class is an `abstract` class. This class contains two abstract methods:

```
public abstract boolean withdraw(double amount);

public abstract String getDescription();
```

5. Create a new Java class, `CheckingAccount`, in the `com.example` package.
   a. `CheckingAccount` should be a subclass of `Account`.

```
public class CheckingAccount extends Account
```

   b. Add an `overDraftLimit` field to the `CheckingAccount` class.

```
private final double overDraftLimit;
```

   c. Add a `CheckingAccount` constructor.

```
public CheckingAccount(double balance, double overDraftLimit) {
    super(balance);
    this.overDraftLimit = overDraftLimit;
}
```

d. Add a `CheckingAccount` constructor that has one parameter.

```
public CheckingAccount(double balance) {
    this(balance, 0);
}
```

e. Override the abstract `getDescription` method inherited from the `Account` class.

```
@Override
public String getDescription() {
    return "Checking Account";
}
```

Note: It is a good practice to add `@Override` to any method that should be overriding a parent class method.

f. Override the abstract `withdraw` method inherited from the `Account` class. The `withdraw` method should allow an account balance to go negative up to the amount specified in the `overDraftLimit` field.

```
@Override
public boolean withdraw(double amount) {
    if(amount <= balance + overDraftLimit) {
        balance -= amount;
        return true;
    } else {
        return false;
    }
}
```

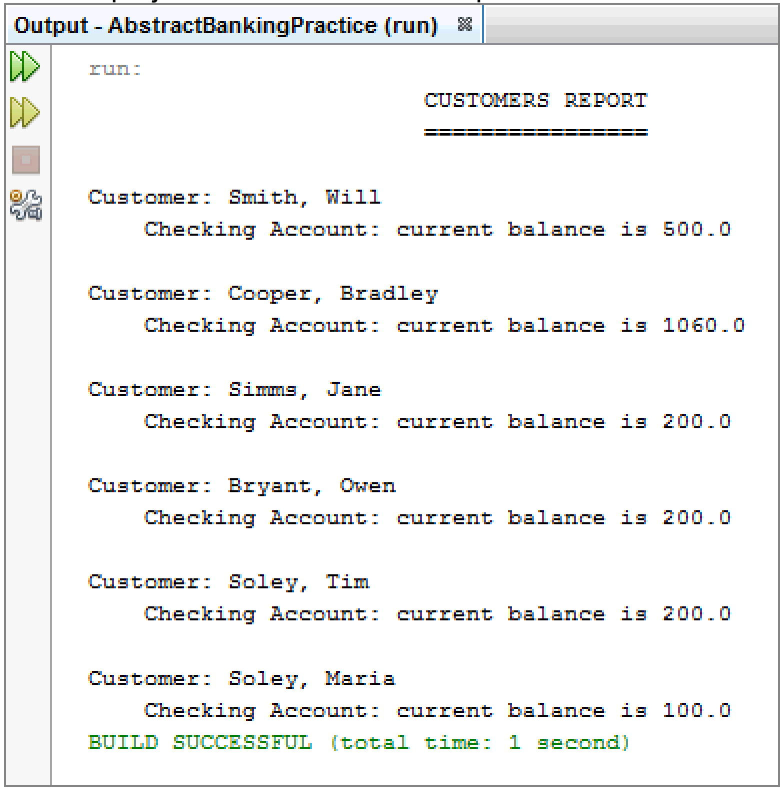
6. Modify the `AbstractBankingMain` class to create checking accounts for the customers.

Note: Both `Customer` and `CustomerReport` can utilize `CheckingAccount` instances, because you previously modified them to use `Account` type references.

```
// Create several customers and their accounts
        bank.addCustomer("Will", "Smith");
        customer = bank.getCustomer(0);
        customer.addAccount(new SavingsAccount(500.00));

        bank.addCustomer("Bradley", "Cooper");
        customer = bank.getCustomer(1);
        SavingsAccount sack = new SavingsAccount(500.00);
        customer.addAccount(sack);
        sack.deposit(500);

        bank.addCustomer("Jane", "Simms");
        customer = bank.getCustomer(2);
        customer.addAccount(new CheckingAccount(200.00,
400.00));
```

```
        bank.addCustomer("Owen", "Bryant");
        customer = bank.getCustomer(3);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Tim", "Soley");
        customer = bank.getCustomer(4);
        customer.addAccount(new CheckingAccount(200.00));


        bank.addCustomer("Maria", "Soley");
        customer = bank.getCustomer(5);
        CheckingAccount chkAcct = new CheckingAccount(100.00);
        customer.addAccount(chkAcct);
        if (chkAcct.withdraw(900.00)) {
            customer.addAccount(chkAcct);
            System.out.print(" withdraw is successful" +
chkAcct.getBalance());
        }
```

7. Run the project. You should see a report of all customers and their accounts.



```
run:
                        CUSTOMERS REPORT
                        ================

Customer: Smith, Will
    Checking Account: current balance is 500.0

Customer: Cooper, Bradley
    Checking Account: current balance is 1060.0

Customer: Simms, Jane
    Checking Account: current balance is 200.0

Customer: Bryant, Owen
    Checking Account: current balance is 200.0

Customer: Soley, Tim
    Checking Account: current balance is 200.0

Customer: Soley, Maria
    Checking Account: current balance is 100.0
BUILD SUCCESSFUL (total time: 1 second)
```

Practice 5-2: Summary Level: Implementing Inner Class as a Helper Class

Overview

In this practice, you will take an existing application and develop an inner class as a helper class to compute employee benefits.

Assumptions

You have reviewed the nested class section of this lesson.

Summary

You have been given a small project that contains an `Employee.java`, implement an inner class as a helper class to compute employee benefits.

Tasks

1. Open the `EmployeeInner05-02Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice2`
   c. Select `EmployeeInner05-02Prac`
   d. Click Open Project.
2. Edit `Employee.java` and make the following changes:
   a. Develop an innerclass, `BenefitsHelper`.
   b. Declare two class variables: `bonusRate` and `withholdingRate`.
   c. Initialize `bonusRate` and `withholdingRate`.

```
private final double bonusRate = 0.02;
private final double withholdingRate = 0.07;
```

   d. Add 2 methods: `calcBonus` (to compute the bonus) and `calcWithholding` (to compute the withhholding).
   e. Create an instance of `BenefitsHelper` in the `Employee` class.
   f. Add 2 getter methods to the `Employee` class to return the bonus and withholding.

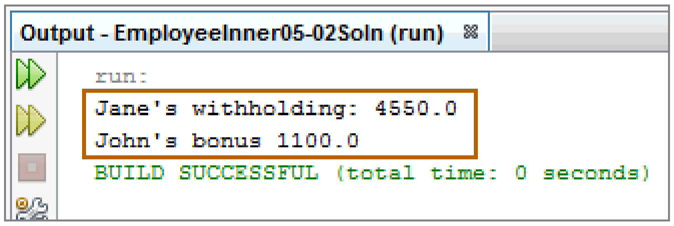
3. Develop `Main.java`:
   a. Create a Java class, `Main.java` in the `com.example` package.
   b. Add a `main` method to the `Main` class.
   c. Perform the following steps in the `main` method:
      - Create two instances of the `Employee` class.

```
Employee jane = new Employee("Jane Doe", "Manager", "HR",
65000);
Employee john = new Employee("John Doe", "Staff", "HR", 55000);
```

- Invoke the `getWithholding()` and `getBonus()` methods to display employee benefits.

```
 System.out.println("Jane's withholding: " +
jane.getWithholding());
 System.out.println("John's bonus " + john.getBonus());
```

4. Run the project. You should see the output in the output window.



```
run:
Jane's withholding: 4550.0
John's bonus 1100.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practice 5-2: Detailed Level: Implementing Inner Class as a Helper Class

Overview

In this practice, you will take an existing application and develop an inner class as a helper class to compute employee benefits.

Assumptions

You have reviewed the nested class section of this lesson.

Summary

You have been given a small project that contains an `Employee.java`, implement an inner class as a helper class to compute employee benefits.

Tasks

1. Open the `EmployeeInner05-02Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice2`
   c. Select `EmployeeInner05-02Prac`
   d. Click Open Project.
2. Expand the project directories.
3. Edit `Employee.java` under the `com.example` package.
   a. Create an inner class, `BenefitsHelper.java` inside the `Employee` class.
   b. Declare two variables: `bonusRate` and `withholdingRate`
   c. Initialize `bonusRate` and `withholdingRate`

```
private final double bonusRate = 0.02;
private final double withholdingRate = 0.07;
```

   d. Add a method `calcBonus` to calculate the bonus of the employee.

```
protected double calcBonus(double salary){
  return salary * bonusRate;
}
```

   e. Add a method `calcWithholding` to calculate the withholding of the employee.

```
protected double calcWithholding(double salary){
    return salary * withholdingRate;
}
```

   f. Create an instance of `BenefitsHelper` in the `Employee` class.

```
private BenefitsHelper helper = new BenefitsHelper();
```

g. Add two getter methods to the `Employee` class to return the bonus and withholding.

   i. Add the `getWithholding()` method:

```
public double getWithholding(){
        return helper.calcWithholding(salary);
    }
```

   ii. Add the `getBonus()` method:

```
 public double getBonus(){
        return helper.calcBonus(salary);
    }
```

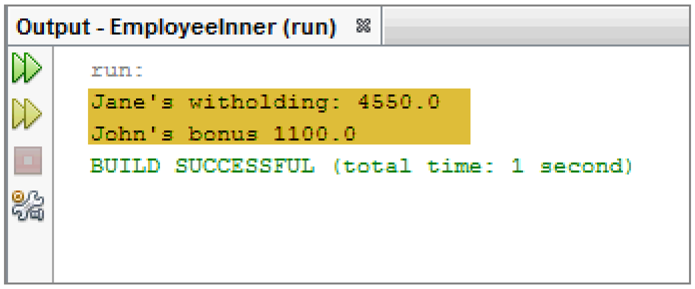
4. Create `Main.java` class under `com.example` package.
5. Modify `Main.java`:
   a. Add a `main` method to the class.
   b. Create 2 instances of the `Employee` class in the `main` method.

```
Employee jane = new Employee("Jane Doe", "Manager", "HR",
65000);
Employee john = new Employee("John Doe", "Staff", "HR", 55000);
```

   c. Invoke the `getWithholding()` and `getBonus()` methods to output the bonus and withholding of the employee instances.

```
System.out.println("Jane's withholding: " +
jane.getWithholding());
System.out.println("John's bonus " + john.getBonus());
```

6. Run the project. You should see the output in the output window.



```
run:
Jane's witholding: 4550.0
John's bonus 1100.0
BUILD SUCCESSFUL (total time: 1 second)
```

Practice 5-3: Summary Level: Using Java Enumerations

Overview

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

Assumptions

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

Summary

You have been given a project that implements the logic for a bank. By creating a new Java enum you will modify the application to hold various branch locations of the bank. By using enum to store the branch details, in the future it is easy to add more branch locations to the bank, it is easy to validate branch information.

Tasks

1. Open the `EnumBanking05-03Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice3`
   c. Select `EnumBanking05-03Prac` and click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Create a new Java enum, `Branch` in the `com.example` package**.**
5. Modify the enum, `Branch.java`. The `Branch enum` stores the location at which the customer banks at. In addition, information about the types of services offered by the bank is also stored.
   a. Create `Branch` instances, `LA, BOSTON, BANGALORE, MUMBAI` that call the `Branch` constructor with values "Basic", "Loan", "Full", and "Full", respectively.
   b. Declare a `serviceLevel` field along with a corresponding constructor and getter method.

```
public enum Branch {

    LA("Basic"), BOSTON("Loan"), BANGALORE("Full"), MUMBAI("Full");

        String serviceLevel;
        private Branch(String serviceLevel){
            this.serviceLevel = serviceLevel;
        }

        public String getServiceLevel(){
          return serviceLevel;
        }


}
```

6. Modify the `Customer` class to store branch information.
   a. Open the `Customer.java` file (under the `com.example` package).
   b. Declare a variable of type Branch.

```
        private Branch branch;
```

   c. Modify the existing constructor to receive an enum, `Branch` as the third parameter.
   d. Add getter and setter methods for the `branch` field.
7. Modify the `Bank` class to modify `addCustomer` method.
   a. Open the `Bank.java` file (under the `com.example` package).
   b. Within the `addCustomer` method, add `Branch` instance as a parameter.
   c. Within the customer instance creation statement, modify the constructor to include `Branch` instance as a parameter.

```
public void addCustomer(String f, String l, Branch b) {
        int i = numberOfCustomers++;
        customers[i] = new Customer(f, l, b);
    }
```

8. Modify the `CustomerReport.java` to display the branch for each customer.

```
            // Print the customer's name
            System.out.println();
            System.out.println("Customer: "
                    + customer.getLastName() + ", "
                    + customer.getFirstName()
                    + "\nBranch: " + customer.getBranch() + ", "
                    + customer.getBranch().getServiceLevel());
```

9. Modify `AbstractBankingMain.java` to update the customers information with the branch details, for example:

```
bank.addCustomer("Will", "Smith",Branch.LA);
        customer = bank.getCustomer(0);
        customer.addAccount(new SavingsAccount(500.00));
```

10. Run the project. You should see a report of all customers and their accounts with the branch locations of the bank.

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

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

Customer: Cooper, Bradley
Branch: BOSTON, Loan
    Checking Account: current balance is 1060.0

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

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

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

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

Practice 5-3: Detailed Level: Using Java Enumerations

Overview

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

Assumptions

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

Summary

You have been given a project that implements the logic for a bank. By creating a new Java enum you will modify the application to hold various branch locations of the bank. By using enum to store the branch details, in the future it is easy to add more branch locations to the bank, it is easy to validate branch information.

Tasks

1. Open the `EnumBanking05-03Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice3`.
   c. Select `EnumBanking05-03Prac`.
   d. Click Open Project.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Create a new Java enum, `Branch` in the `com.example` package, by performing the following steps:
   a. In NetBeans, right-click on the project, select New > Other.
   b. Select **Java** from Categories column
   c. Select **Java Enum** from File Types column
   d. Click **Next.**
5. In the Name and Location dialog box, enter the following details:
   a. Class: `Branch`
   b. Package: `com.example`
   c. Click Finish.
6. Modify the enum, `Branch.java`. The `Branch` `enum` stores the location at which the customer banks at. In addition, information about the types of services offered by the bank are also stored.
   a. Create `Branch` instances, `LA, BOSTON, BANGALORE, MUMBAI` that call the `Branch` constructor with values "Basic", "Loan", "Full", and "Full", respectively.
   b. Declare a `serviceLevel` field along with a corresponding constructor and getter method.

```
public enum Branch {

    LA("Basic"), BOSTON("Loan"), BANGALORE("Full"), MUMBAI("Full");
```

```
        String serviceLevel;
        private Branch(String serviceLevel){
            this.serviceLevel = serviceLevel;
        }

        public String getServiceLevel(){
          return serviceLevel;
        }


}
```

7. Modify the `Customer` class to store branch information.
   a. Open the `Customer.java` file (under the `com.example` package).
   b. Declare a variable of type Branch.

```
      private Branch branch;
```

   c. Modify the existing constructor to receive an enum, `Branch` as the third parameter.

```
    public Customer(String f, String l,Branch b) {
        firstName = f;
        lastName = l;
        // initialize accounts array
        accounts = new Account[10];
        numberOfAccounts = 0;
        branch=b;

    }
```

   d. Add getter and setter methods for the `branch` field.

```
public Branch getBranch() {
        return branch;
    }

    public void setBranch(Branch branch) {
        this.branch = branch;
    }
```

8. Modify the `Bank` class to modify `addCustomer` method.
   a. Open the `Bank.java` file (under the `com.example` package).
   b. Within the `addCustomer` method, add `Branch` instance as a parameter.
   c. Within the customer instance creation statement, modify the constructor to include `Branch` instance as a parameter.

```
public void addCustomer(String f, String l, Branch b) {
        int i = numberOfCustomers++;
        customers[i] = new Customer(f, l, b);
    }
```

9. Modify the `CustomerReport.java` to display the branch for each customer.

```
            // Print the customer's name
            System.out.println();
            System.out.println("Customer: "
                    + customer.getLastName() + ", "
                    + customer.getFirstName()
                    + "\nBranch: " + customer.getBranch() + ", "
                    + customer.getBranch().getServiceLevel());
```

10. Modify `AbstractBankingMain.java` to update the customer's information with the branch details.

```
bank.addCustomer("Will", "Smith",Branch.LA);
        customer = bank.getCustomer(0);
        customer.addAccount(new SavingsAccount(500.00));

        bank.addCustomer("Bradley", "Cooper", Branch.BOSTON);
        customer = bank.getCustomer(1);
        SavingsAccount sack = new SavingsAccount(500.00);
        customer.addAccount(sack);
        sack.deposit(500);

        bank.addCustomer("Jane", "Simms", Branch.MUMBAI);
        customer = bank.getCustomer(2);
        customer.addAccount(new CheckingAccount(200.00, 400.00));

        bank.addCustomer("Owen", "Bryant", Branch.BANGALORE);
        customer = bank.getCustomer(3);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Tim", "Soley", Branch.LA);
        customer = bank.getCustomer(4);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Maria", "Soley",Branch.BANGALORE);
        customer = bank.getCustomer(5);
        CheckingAccount chkAcct = new CheckingAccount(100.00);
```

d. Run the project. You should see a report of all customers and their accounts with the branch locations of the bank.

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

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

Customer: Cooper, Bradley
Branch: BOSTON, Loan
    Checking Account: current balance is 1060.0

Customer: Simms, Jane
Branch: MUMBAI, Full
```

```
    Checking Account: current balance is 200.0

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

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

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