Practices for Lesson 4

Practices Overview

In these practices, you will use the abstract, final, and static Java keywords. You will also learn to recognize nested classes.

Practice 4-1: Summary 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 time deposit accounts. Time deposit accounts allow withdraw only after a maturity date. Time deposit accounts are also known as term deposit, certificate of deposit (CD), or fixed deposit accounts. You will enhance the software to support checking accounts.

A checking account and a time deposit account have some similarities and some differences. Your class design should reflect this. Additional types of accounts might be added in the future.

Tasks

1. Open the `AbstractBanking` project as the main project.
    a. Select File > Open Project.
    b. Browse to `D:\labs\04\practices (or your other directory).`
    c. Select `AbstractBanking` 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 a report of all customers and their accounts.
4. Review the `TimeDepositAccount` class.
    a. Open the `TimeDepositAccount.java` file (under the `com.example` package).
    b. Identify the fields and method implementations of `TimeDepositAccount` that are related to time or are in some other way specific to `TimeDepositAccount`. Add a code comment if desired.
    c. Identify the fields and method implementations of `TimeDepositAccount` that could be used by any type of account. Add a code comment if desired.
5. Create a new Java class, `Account`, in the `com.example` package.
6. Code the `Account` class.
    a. This class should be declared as `abstract`.
    b. Move any fields and method implementations from `TimeDepositAccount` that could be used by any type of account to the `Account` class.

       Note: The fields and methods should be removed from `TimeDepositAccount`.

   c. Add `abstract` methods to the `Account` class for any methods in `TimeDepositAccount` that are time related but have a method signature that would make sense in any type of account.

      Hint: Would all accounts have a description?

   d. Add an `Account` class constructor that has a `double balance` parameter.

   e. The `Account` class should have a protected access level `balance` field that is initialized by the `Account` constructor.

7. Modify the `TimeDepositAccount` class.

   a. `TimeDepositAccount` should be a subclass of `Account`.

   b. Modify the `TimeDepositAccount` constructor to call the parent class constructor with the `balance` value.

   c. Make sure that you are overriding the abstract `withdraw` and `getDescription` methods inherited from the `Account` class.

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

8. Modify the `Customer` and `CustomerReport` classes to use `Account` references.

   a. Open the `Customer.java` file (under the `com.example` package).

   b. Change all `TimeDepositAccount` references to `Account` type references.

   c. Open the `CustomerReport.java` file (under the `com.example` package).

   d. Change all `TimeDepositAccount` references to `Account` type references.

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

10. 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 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.
- The `withdraw` method should return `false` if the withdraw cannot be performed, and `true` if it can.

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

```
// Create several customers and their accounts
bank.addCustomer("Jane", "Simms");
customer = bank.getCustomer(0);
customer.addAccount(new TimeDepositAccount(500.00,
cal.getTime()));
customer.addAccount(new CheckingAccount(200.00, 400.00));

bank.addCustomer("Owen", "Bryant");
customer = bank.getCustomer(1);
customer.addAccount(new CheckingAccount(200.00));

bank.addCustomer("Tim", "Soley");
customer = bank.getCustomer(2);
customer.addAccount(new TimeDepositAccount(1500.00,
cal.getTime()));
customer.addAccount(new CheckingAccount(200.00));

bank.addCustomer("Maria", "Soley");
customer = bank.getCustomer(3);
// Maria and Tim have a shared checking account
customer.addAccount(bank.getCustomer(2).getAccount(1));
customer.addAccount(new TimeDepositAccount(150.00,
cal.getTime()));
```

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

12. Run the project. You should see a report of all customers and their accounts. Note that the date displayed should be one hundred and eighty days in the future.

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

Customer: Simms, Jane
    Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 500.0
    Checking Account: current balance is 200.0

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

Customer: Soley, Tim
    Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 1500.0
    Checking Account: current balance is 200.0

Customer: Soley, Maria
    Checking Account: current balance is 200.0
    Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 150.0
```

Practice 4-1: 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 time deposit accounts. Time deposit accounts allow withdraw only after a maturity date. Time deposit accounts are also known as term deposit, certificate of deposit (CD), or fixed deposit accounts. You will enhance the software to support checking accounts.

A checking account and a time deposit account have some similarities and some differences. Your class design should reflect this. Additional types of accounts might be added in the future.

Tasks

1. Open the `AbstractBanking` project as the main project.
    a. Select File > Open Project.
    b. Browse to `D:\labs\04\practices`. (or your other directory)
    c. Select `AbstractBanking` 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 a report of all customers and their accounts.

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

Customer: Simms, Jane
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 500.0

Customer: Bryant, Owen

Customer: Soley, Tim
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 1500.0

Customer: Soley, Maria
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 150.0
```

4. Review the `TimeDepositAccount` class.
    a. Open the `TimeDepositAccount.java` file (under the `com.example` package).
    b. Identify the fields and method implementations of `TimeDepositAccount` that are related to time or are in some other way specific to `TimeDepositAccount`. Add a code comment to the `maturityDate` field and the `withdraw` and `getDescription` methods. For example:

```
// time deposit account specific code
private final Date maturityDate;
```

    c. Identify the fields and method implementations of `TimeDepositAccount` that could be used by any type of account. Add a code comment to the `balance` field and the `getBalance`, `deposit`, and `toString` methods. For example:

```
// generic account code
private double balance;
```

5. Create a new Java class, `Account`, in the `com.example` package.
6. Code the `Account` class.
    a. This class should be declared as `abstract`.

```
public abstract class Account
```

    b. Move the `balance` field and the `getBalance`, `deposit`, and `toString` methods from `TimeDepositAccount` to the `Account` class.

    Note: The fields and methods should be removed from `TimeDepositAccount`.

    c. Add an abstract `getDescription` method to the `Account` class.

```
public abstract String getDescription();
```

    d. Add an abstract `withdraw` method to the `Account` class.

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

    e. The `Account` class should have a protected access level `balance` field. If you have already moved this field from the `TimeDepositAccount`, just change the access level.

```
protected double balance;
```

    f. Add an `Account` class constructor that has a `double balance` parameter.

```
public Account(double balance) {
    this.balance = balance;
}
```

7. Modify the `TimeDepositAccount` class.
    a. `TimeDepositAccount` should be a subclass of `Account`.

```
public class TimeDepositAccount extends Account
```

b. Modify the `TimeDepositAccount` constructor to call the parent class constructor with the `balance` value.

```
super(balance);
```

c. Make sure that you are overriding the abstract `withdraw` and `getDescription` methods inherited from the `Account` class, by using the `@Override` annotation.

```
@Override
public String getDescription() {
    return "Time Deposit Account " + maturityDate;
}
```

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

8. Modify the `Customer` and `CustomerReport` classes to use `Account` references.
   a. Open the `Customer.java` file (under the `com.example` package).
   b. Change all `TimeDepositAccount` references to `Account` type references.
   c. Open the `CustomerReport.java` file (under the `com.example` package).
   d. Change all `TimeDepositAccount` references to `Account` type references.
9. Run the project. You should see a report of all customers and their accounts.
10. 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;
    }
}
```

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

```
// Create several customers and their accounts
bank.addCustomer("Jane", "Simms");
customer = bank.getCustomer(0);
customer.addAccount(new TimeDepositAccount(500.00,
cal.getTime()));
customer.addAccount(new CheckingAccount(200.00, 400.00));

bank.addCustomer("Owen", "Bryant");
customer = bank.getCustomer(1);
customer.addAccount(new CheckingAccount(200.00));

bank.addCustomer("Tim", "Soley");
customer = bank.getCustomer(2);
customer.addAccount(new TimeDepositAccount(1500.00,
cal.getTime()));
customer.addAccount(new CheckingAccount(200.00));

bank.addCustomer("Maria", "Soley");
customer = bank.getCustomer(3);
// Maria and Tim have a shared checking account
customer.addAccount(bank.getCustomer(2).getAccount(1));
customer.addAccount(new TimeDepositAccount(150.00,
cal.getTime()));
```

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

12. Run the project. You should see a report of all customers and their accounts. Note that the date displayed should be one hundred and eighty days in the future.

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


Customer: Simms, Jane
    Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 500.0
    Checking Account: current balance is 200.0


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


Customer: Soley, Tim
    Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 1500.0
    Checking Account: current balance is 200.0


Customer: Soley, Maria
    Checking Account: current balance is 200.0
    Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 150.0
```

Practice 4-2: Summary Level: Applying the Singleton Design Pattern

Overview

In this practice, you will take an existing application and refactor the code to implement the Singleton design pattern.

Assumptions

You have reviewed the static and final keyword sections of this lesson.

Summary

You have been given a project that implements the logic for a bank. The application currently allows the creation of an unlimited number of `Bank` instances.

```
Bank bank = new Bank();
Bank bank2 = new Bank();
Bank bank3 = new Bank();
```

Using the static and final keywords you will limit the number of `Bank` instances to one per Java virtual machine (JVM).

Tasks

1. Open the `SingletonBanking` project as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\05\practices (or your other directory)`.
   c. Select `SingletonBanking` 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 a report of all customers and their accounts.

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


Customer: Simms, Jane
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 500.0

Customer: Bryant, Owen

Customer: Soley, Tim
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 1500.0

Customer: Soley, Maria
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 150.0
```

4. Modify the `Bank` class to implement the Singleton design pattern.
   a. Open the `Bank.java` file (under the `com.example` package).
   b. Change the constructor's access level to `private`.
   c. Add a new field named `instance`. The field should be:
      - `private`
      - Marked `static`
      - Marked `final`
      - Type of `Bank`
      - Initialized to a new `Bank` instance
   d. Create a static method named `getInstance` that returns the value stored in the `instance` field.
5. Modify the `SingletonBankingMain` class to use the `Bank` singleton.
   a. Open the `SingletonBankingMain.java` file (under the `com.example` package).
   b. Replace any calls to the Bank constructor with calls to the previously created `getInstance` method.
   c. In the `main` method, create a second local `Bank` reference named `bank2` and initialize it using the `getInstance` method.
   d. Use reference equality checking to determine whether bank and bank2 reference the same object.

      ```
      if(bank == bank2) {
          System.out.println("bank and bank2 are the same object");
      }
      ```

   e. Try initializing only the second Bank but running the report on the first Bank.

      ```
      initializeCustomers(bank2);

      // run the customer report
      CustomerReport report = new CustomerReport();
      report.setBank(bank);
      report.generateReport();
      ```

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

Practice 4-2: Detailed Level: Applying the Singleton Design Pattern

Overview

In this practice, you will take an existing application and refactor the code to implement the Singleton design pattern.

Assumptions

You have reviewed the static and final keyword sections of this lesson.

Summary

You have been given a project that implements the logic for a bank. The application currently allows the creation of an unlimited number of `Bank` instances.

```
Bank bank = new Bank();
Bank bank2 = new Bank();
Bank bank3 = new Bank();
```

Using the static and final keywords you will limit the number of `Bank` instances to one per Java Virtual Machine (JVM).

Tasks

1. Open the `SingletonBanking` project as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\04\practices`. (or your other directory)
   c. Select `SingletonBanking` 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 a report of all customers and their accounts.
4. Modify the `Bank` class to implement the Singleton design pattern.
   a. Open the `Bank.java` file (under the `com.example` package).
   b. Change the constructor’s access level to `private`.

```
private Bank() {
    customers = new Customer[10];
    numberOfCustomers = 0;
}
```

   c.

Add a new field named `instance`. The field should be:

- `private`
- Marked `static`
- Marked `final`
- Type of `Bank`
- Initialized to a new `Bank` instance

```
private static final Bank instance = new Bank();
```

d. Create a static method named `getInstance` that returns the value stored in the `instance` field.

```
public static Bank getInstance() {
    return instance;
}
```

5. Modify the `SingletonBankingMain` class to use the `Bank` singleton.

   a. Open the `SingletonBankingMain.java` file (under the `com.example` package).

   b. Replace any calls to the Bank constructor with calls to the previously created `getInstance` method.

   ```
   Bank bank = Bank.getInstance();
   ```

   c. In the `main` method, create a second local `Bank` reference named `bank2` and initialize it using the `getInstance` method.

   ```
   Bank bank2 = Bank.getInstance();
   ```

   d. Use reference equality checking to determine whether bank and bank2 reference the same object.

   ```
   if(bank == bank2) {
       System.out.println("bank and bank2 are the same object");
   }
   ```

   e. Initialize only the second Bank, but run the report on the first Bank.

   ```
   initializeCustomers(bank2);

   // run the customer report
   CustomerReport report = new CustomerReport();
   report.setBank(bank);
   report.generateReport();
   ```

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

(Optional) Practice 4-3: 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. The application currently allows the creation of `TimeDepositAccount` instances with any maturity date.

```
//180 day term
Calendar cal = Calendar.getInstance();
cal.add(Calendar.DAY_OF_YEAR, 180);
new TimeDepositAccount(500.00, cal.getTime())
```

By creating a new Java enum you will modify the application to only allow for the creation of `TimeDepositAccount` instances with a maturity date that is 90 or 180 in the future.

Tasks

1. Open the `EnumBanking` project as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\04\practices (or your other directory).`
   c. Select `EnumBanking` 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 a report of all customers and their accounts.
4. Create a new Java enum, `DepositLength`, in the `com.example` package.
5. Code the `DepositLength` enum.
   a. Declare a `days` field along with a corresponding constructor and getter method.

```
private int days;

private DepositLength(int days) {
    this.days = days;
}

public int getDays() {
    return days;
}
```

   b. Create two `DepositLength` instances, `THREE_MONTHS` and `SIX_MONTHS` that call the `DepositLength` constructor with values of 90 and 180 respectively.

6. Modify the `TimeDepositAccount` class to only accept `DepositLength` instances for the constructor's maturity date parameter.
    a. Open the `TimeDepositAccount.java` file (under the `com.example` package).
    b. Modify the existing constructor to receive an enum as the second parameter.

```
public TimeDepositAccount(double balance, DepositLength
duration) {
    super(balance);
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.DAY_OF_YEAR, duration.getDays());
    this.maturityDate = cal.getTime();
}
```

7. Modify the `EnumBankingMain` class to create `TimeDepositAccount` instances using the two `DepositLength` instances available.
    a. Open the `EnumBankingMain.java` file (under the `com.example` package).
    b. Within the `initializeCustomers` method, remove the code to create calendars.
    c. Within the `initializeCustomers` method, modify the creation of all `TimeDepositAccount` instances to use the `DepositLength` enum.

```
customer.addAccount(new TimeDepositAccount(500.00,
DepositLength.SIX_MONTHS));
```

Note: Try using both the `SIX_MONTHS` and `THREE_MONTHS` values. You can also use a static import to reduce the length of the statement.

8. Run the project. You should see a report of all customers and their accounts. It is now impossible to compile a line of code that creates a `TimeDepositAccount` with an invalid maturity date.

(Optional) Practice 4-4: Recognizing Nested Classes

Overview

In this practice, you will take an existing application and attempt to recognize the declaration and use of various types of nested classes.

Assumptions

You have reviewed the nested class section of this lesson.

Summary

You have been given a small project that contains only two `.java` files. Although there are only two `.java` files, there may be multiple Java classes being created.

Attempt to determine the number of classes being created.

Tasks

1. Open the `NestedClasses` project as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\04\practices`. (or your other directory)
   c. Select `NestedClasses` 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 the output in the output window.
4. Count the number of classes created in the `OuterClass.java` file.
   a. Open the `OuterClass.java` file (under the `com.example` package).
   b. Determine the total number of classes created in this file.
   c. Determine the total number of top-level classes created in this file.
   d. Determine the total number of nested classes created in this file.
   e. Determine the total number of inner classes.
   f. Determine the total number of member classes.
   g. Determine the total number of local classes.
   h. Determine the total number of anonymous classes.
   i. Determine the total number of static nested classes.

   Hint: Using the Files tab in NetBeans, you can see how many `.class` files are created by looking in the `build\classes` folder for a project.

(Optional) Solution 4-4: Recognizing Nested Classes

Overview

In this solution, you will take an existing application and review the number and types of nested classes created within a single `.java` file.

Assumptions

You have reviewed the nested class section of this lesson.

Summary

You have been given a small project that contains only two `.java` files. Although there are only two `.java` files, there may be multiple Java classes being created.

Review the number of classes being created.

Tasks

1. Open the `NestedClasses` project as the main project.
    a. Select File > Open Project.
    b. Browse to `D:\labs\04\practices (or your other directory).`
    c. Select `NestedClasses` 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 the output in the output window.
4. Open the `OuterClass.java` file (under the `com.example` package).
    - Within the `OuterClass.java` file there are:
        - 10 classes
            - 1 top-level class
            - 9 nested classes
                - 8 inner classes
                    - 3 member classes
                    - 2 local classes
                    - 3 anonymous classes
                - 1 static nested class

- Classes are declared on the following lines within the `OuterClasss.java` file:
  - line 3: top-level class
  - line 10: local inner class
  - line 22: anonymous local inner class
  - line 32: anonymous inner class
  - line 40: anonymous inner class
  - line 48: member inner class
  - line 62: static nested class
  - line 72: member inner class
  - line 74: member inner class
  - line 77: local inner class

  Hint: You can show line number in NetBeans by going to the View menu and enabling Show Line Numbers.