SCHOOL OF COMPUTER SCIENCE

PRIFYSGOL
BANGOR
UNIVERSITY

Object Oriented Programming in Java

Laboratory 4
Inheritance

Dave Perkins

Introduction

This laboratory session explores the use of the *inheritance relationship* in Java; this will involve creating both superclasses and subclasses. Remember that a subclass *inherits* all methods from the superclass which have been declared as public. The subclass may also add *new* instance variables and methods or perhaps redefine existing methods (known as overriding).

Inheritance supports the ideal of *software reuse* and is central to the object oriented approach to programming.

Try to complete as many exercises as possible. As usual make sure that you save all your laboratory work in an appropriate folder.

Exercise 1: The Bank Hierarchy

The following exercises (1-5) are based on the **`BankAccount`** class introduced in Lecture 3.

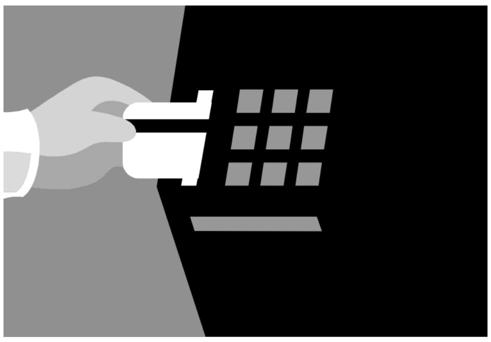

Figure 1. Implementing a Bank Account

if you have not already done so create the following classes from Horstmann's *Big Java*:

```
•  BankAccount          // Superclass
•  SavingAccout         // Subclass
•  CheckingAccount      // Subclass
```

Each of these classes should have appropriate **`toString`** and **`format`** methods. Consult Horstmann to see how to write a **`toString`** class for a subclass.

Once the hierarchy has been created develop the following test classes:

- **`BankAccountTester`**
- **`SavingsAccountTester`**
- **`CheckingAccountTester`**

Consult the early Semester 2 lectures for examples of tester classes.

In developing the test classes follow the guidelines below:

- Test all constructors;
- Test every method available to an instance of a class;
- Insert println statements to display expected and actual values;
- Use toString to display the state of an object.

Finally, make sure that you have included Javadoc style comments wherever appropriate.

Exercise 2: Polymorphic Methods

Provide **`BankAccount`** with a **`transfer`** method as discussed in Lectyre 5. Amend the test classes as required.

Figure 2. Transferring Between Accounts

Exercise 3: Modifying Savings Account

Modify the **`addInterest`** method of **`SavingsAccount`** to compute the interest on the *minimum balance* since the last call to **`addInterest`**. After the calculation has been made the minimum balance should be reset to equal the current balance. **Hint**: you will need to produce a modified version of the **`withdraw`** method for use in the **`SavingsAccount`** class. This will require an instance variable to remember the minimum balance. Thus every time a withdrawal is made from a savings account the minimum balance field is checked and if the new balance is below the minimum then the latter is updated. tester class.

Exercise 4: Providing a New Class

Add a **`TimedDepositAccount`** class to the bank account hierarchy. The time deposit account is just like a savings account except that the customer must leave their money in the account for a specific number of months, and there is a fixed $20 penalty for early withdrawal. Construct the account with the interest rate and the number of months to maturity. In the **`addInterest`** method, decrement the count of months. If the count is positive during a withdrawal, charge the withdrawal penalty. Provide an appropriate new tester class.

Exercise 5: Abstract and Concrete

Reorganize the bank account classes as follows. In the **BankAccount** class, introduce an abstract method **endOfMonth** with no implementation. Rename the **addInterest** and **deductFees** methods as **endOfMonth** in the relevant subclasses. Which classes are now abstract and which are concrete? Write a test class that makes five transactions and then calls **endOfMonth**. Test using instances of all concrete bank account classes.

Exercise 6: A New Class Hierarchy

Develop a class called **Book** so that a program can represent information about books. Our initial assumption will be that a book is defined by just two instance variables – the number of pages it contains and its title. Provide two access methods called **getNoPages()** and **getTitle()** for this class as well as corresponding setter methods.

Also do not forget to provide an appropriate constructor and a **toString** method

Having created this class develop a test driver **BookTester** and confirm that you can create instances of the class and extract information regarding a book's title and length.

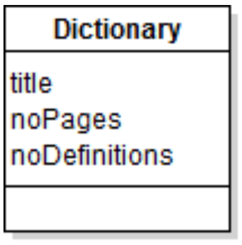

Figure 3 UML Diagram for Class Dictionary

Now define a subclass of **Book** called **Dictionary**. The distinctive feature of a dictionary is that it contains definitions and this fact must be reflected in your subclass. Thus **Dictionary** will contain

- a new *instance* variable called **noDefinitions** to record how many definitions an individual dictionary contains;
- an access method for this variable.

Once again do not forget to provide a constructor for the new class.

As with **Book** develop a test driver for this new class.

Exercise 7: The Worker Classes

Develop a superclass called **`Worker`** and two subclasses – **`HourlyWorker`** and **`SalariedWorker`**; in developing this class hierarchy you should take note of the following:

- every worker has a name and a basic pay rate (i.e. hourly rate);
- the weekly pay for an hourly paid worker is equal to the pay rate * the number of hours actually worked;
- if an hourly paid worker works more than 40 hours then they are paid at time and a half;
- the weekly pay for a salaried worker is 40 * the basic pay rate.

Provide *appropriate* get and set methods for all three of the classes mentioned above. Also provide one or more versions of the method :

```
double computePay(int hours)
```

This method calculates and returns the weekly pay for any type of worker.

Finally provide a **`toString`** method for every class in the hierarchy. If you have forgotten read *Big Java:Late Objects*, Chapter 9 (pp.446-447) on how to override the **`toString`** method.

Exercise 8: All at Sea

Develop a superclass called **`Ship`** with the following components:

- an instance variable to hold the name of the ship;
- an instance variable to hold the year the ship was built;
- one or more constructors ;
- all necessary access and mutator methods;
- a **`toString`** method that displays the ship's name and the year it was built.

With regard to the number of constructors for this class the choice is yours. You may supply one or more but be prepared to explain your design decision.

Develop a subclass **`CruiseShip`** that extends the **`Ship`** class. This subclass class has the following components

- an instance variable to hold the maximum number of passengers;
- one or more constructors;
- all necessary access and mutator methods;
- a **`toString`** method that overides the **`toString`** method in the base class and which displays the ship's name and maximum number of passengers.

Develop a subclass **`CargoShip`** that extends the **`Ship`** class. The new class should contain the following features:

- an instance variable for the cargo capacity in tonnage;
- one or more constructors;
- all necessary access and constructor methods;
- a **`toString`** method that overides the **`toString`** method in the base class and which displays the ship's name and the cargo capacity.

Finally, develop a test driver to demonstrate the functionality of all of these classes.

Exercise 9: Ships Ahoy!

Create an array called **`fleet`** to hold a fleet of five ships. Create two cargo ships and three cruise ships and put them into the array. Now use a **`for`** loop to print out the details of each of the ships in the array.

Insert comments to identify instances of polymorphic behaviour in your program.

****