**chap 4 Finish the crab game**

**Objectives**

* Learn how to create object use the keyword new, and keyword super.
* Learn how to create and assign values to primitive type variables, and to objects.
* Understand that objects are class type variables.
* Understand the difference between instance variable in a class and local variable in a method
* Understand that assignment operator = is different from equal comparator ==
* Learn how to use image and how to use constructor
* Learn how to create an OOP class in Eclipse, and create an application class to test the OOP class
* Learn how to draw UML class diagram for OOP class, and export UML class diagram to an image file

**Schedule:** This lesson covers Moodle folder “chap 4”. Please follow the steps below.

1, work on sec 4.1 ~ 4.15, and finish the little crab game from exercise 4.1 to 4.26. During the process, you need to know how to create object and use object, and apply images to your game. Scenarios covered and used in this chapter: little-crab-4 and little-crab-5,

2,pay attention to the two pictures on top of page 56 and 57 of the textbook, which indicate that primitive type data holds the value of the variable, while class type data, which is object, holds the reference of the body of the object, and the body of the object locates in the heap. These two pictures are in accordance of figure 2 in file “MemoryManagementForObject.docx” that we just discussed in Moodle folder “chap 3”. Please review this WORD document in “chap 3”, and make sure you understand how primitive and class type data are stored differently in memory.

3, work on sec 4.16 to learn the new ideas about expanding the little crab game.

4, work on textbook exercises 4.35, and learn how to create a scenario from scratch, and how to customize the subclass of the class World. You also need to invoke APIs in greenfoot to detect whether the Ball class object has reached any edge or not.

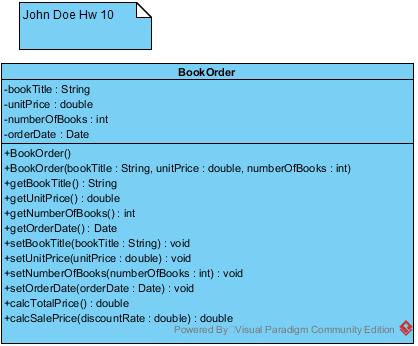
5, Download and unzip file “UML-classDiagram.zip” in Moodle folder “chap 4”, then finish the following tasks:

* View file “CorrectSequence.java” with a source editor such as Notepad++ (in Windows) or BBEdit(in Mac, and understand the correct sequence of writing an class with OOP principles from scratch.
* Create an Eclipse Java project, and include file ***BookOrder.java*** and ***TestBookOrder.java*** into this project. Please refer to file “*CompileAndRunTheFirstJavaProgram.docx*” in Moodle folder “chap 1” on how to create an Eclipse project and how to include existing java files into the project. In file “CompileAndRunTheFirstJavaProgram.docx”, I explain how to include an existing java source code file into Eclipse project. And in the example here, you need to include two source code files “BookOrder.java” and “TestBookOrder.java”, and the approach is the same: you just repeat the drag-and-drop process twice, and place these two files into the “src” folder of the Eclipse project.
* Read file BookOrder.java, which is an example of the correct sequence of writing an OOP class, and cross reference BookOrder.java with CorrectSequence.java. We call BookOrder the ***OOP class***, or the class obeys the encapsulation principle.
* Read file TestBookOrder.java, which is an application file that makes use of the OOP class: BookOrder. We call class TestBookOrder the ***user class***, or ***application class***, which uses the class BookOrder.
* Follow the comments in line 84~95 in file TestBookOrder, and uncomment line 83 and see what happen in the Eclipse compiler error message. This exercise will help you better understand the encapsulation principle of OOP.
* Follow the requirement in file “DownloadAndUseVisualParadigmUML.docx”, install and activate a community edition of Visual Paradigm, and draw the UML class diagram for class BookOrder.java. After finishing the diagram, export the UML class diagram into a jpg file. In UML class diagram, private data members start with – minus sign, while public method members starts with + plus sign. The syntax for representing a data is

dataName : datatype

, and you can see , there is a colon : symbol separating the data name from the data type.

UML stands for Universal Modeling Language, and from the name we can tell it is not specifically designed for any particular programming language such Java, and instead, it can be used under universal platforms. UML class diagram is a very useful tool in OOP design, and you need to learn how to draw UML class diagram. UML class diagram “BookOrder.jpg” is included as below:



6, Some further explanations of encapsulation principle in BookOrder and TestBookOrder:

* The data members (bookTitle, unitPrice, numberOfBooks, orderDate) in class BookOrder are defined as private from line 17 to 20, therefore, they are not visible by any class outside BookOrder.
* In other words, the private data members of class BookOrder are only visible inside class BookOrder, and all methods inside class BookOrder can use these private data members directly without any hesitation. For example, you can refer to line 76 in class BookOrder, and see how two private data members (unitPrice and numberOfBooks) are access directly in this line, and inside this method calTotalPrice().
* The method members (constructors, getters, setters, and effectors) in class BookOrder are defined as public, therefore, they are visible by any class outside BookOrder, which implies that, any other class can call these methods directly without any hesitation, provided that the correct parameter(s) are supplied, if there is any parameter required by these methods.
* The public method members in class BookOrder are also visible to each other inside class BookOrder. Refer to line 82 in class BookOrder, and see how one method ***calcSalePric*e()** calls another method ***calTotalPrice***() of the same class.
* For class TestBookOrder, in its main method, if it wants to access the private data members of another class, for example class BookOrder, then class TestBookOrder has to rely on the public method defined in class BookOrder to access the private data of class BookOrder. For example: in line 83 of TestBookOrder, when it tries to access other class’s private data member like ***myOrder.unitPrice***, it won’t compile, and it has to be changed to line 87, such as ***myOrder.getUnitPrice()***, to make the source code compile. Similar examples can also be found in file TestBookOrder.java:
  + in line 66, where you need to call the public setter to change the private data value of class BookOrder; and
  + in line 72, where you need to call public methods to obtain the values of the private data members of BookOrder (book title, order date), and call public effector to calculate a derived value (the total price, which is not a private data member of class BookOrder) ; and
  + in line 79, where you need to call public effector with plugged-in parameter, in order to obtain a derived value (the sale price, which is not a private data member of class BookOrder)
* The syntax of indirectly accessing other class’s private data or a derived value is as below:

***objectName.publicMethodName***(parameters if needed);

, where the objectName is an object we have created for the other class. For example, in class TestBookOrder, the other class is BookOrder, we create an object of BookOrder and name it as myOrder, and then call ***myOrder.calcSalePrice(discountPercent)*** in line 46, ***or myOrder.getBookTitle()*** in line 39. Similarly we create a second object of class BookOrder and name it as hisOrder, then we call ***hisOrder.setBookTitle(hisBooktitle)*** in line 66, or ***hisOrder.calTotalPrice()*** in line 72.

* For java keyword ***this***, which is used in line 25 and line 30 besides other lines of file BookOrder, please refer to page 14 in file “Test1StudyGuide.docx”, on the two different usages of keyword ***this*** .
* The syntax of accessing a private data member in the same class has two different forms:
  + Access it directly, without the need to call getter method. Please refer to line 76 and the comments in line 76 and 77 in file BookOrder.java.
  + Access it using java keyword “***this***.privateDataName”, such as in line 30~32, line 60, 65, and 70 in file BookOrder.java, because in these lines, there is a formal parameter that has the same name as the private data member coexisting in the method, thus we have to use keyword ***this*** to distinguish the private data member from the formal parameter. For example, in line 65, we have:

***this***.*unitPrice* = ***unitPrice***;

, and the *unitPrice* on the right hand side of the assignment operator = is the formal parameter that is defined in line 64; while the ***this***.*unitPrice* on the left hand side of the assignment operator = is the private data member of class BookOrder defined in line 18. Without the java keyword this in line 65, you cannot distinguish the formal parameter from the private data member in line 65 of method setUnitPrice().

* + If we change the name of the formal parameter from ***unitPrice*** to ***price*** in line 64, then line 65 can be changed to

***unitPrice*** = ***price***;

, and there is no need to use Java keyword ***this***, because the formal parameter ***price*** has a different name than the private data member’s name ***unitPrice*** , so these two variables can show up together in one method without causing any ambiguity. The comments from line 64 to 66 in file BookOrder.java also explain when we can do without java keyword ***this***.

* If we associate these two classes with the analogy in slide #9 and #10 in file “OOP-BasicIntro.ppt” in Moodle folder “chap 2”, then class BookOrder is comparable to a car, which encapsulates private data with public handlers(methods), and class TestBookOrder is comparable to the driver of the car, which uses the public handlers (methods) provided in the car. Please make sure you understand this analogy, which can facilitate your understanding of encapsulation principle. Also, be aware of the two different roles of the two classes: class BookOrder is the OOP class being used (the car being driven), and class TestBookOrder is the user class or application class that uses the OOP class (the driver who drives the car).
* **The first aspect** of the encapsulation principle of OOP: let’s assume there is a class named A, then the private data members of class A are ***encapsulated/protected/hidden*** by the public methods of class A, so other classes cannot directly access the private data in class A, therefore, these other class have to rely on the public methods provided in class A, in order to indirectly access the private data of class A.
* **The second aspect** of the encapsulation principle of OOP: still assume there is a class named A, and for the private data members defined in class A, the public methods defined in class A have direct access to all the private data members defined in the same class A. The comments in line 76 & 77 of file “BookOrder.java” explain this. In addition, the comments in line 74 and 75 of file “BookOrder.java” explains that, there is no need to add a formal parameter to a method, if the data member that needs to be used in the method is already defined as a private data in this class.
* **An example of the second aspect of encapsulation principle:**

The method in line 75 of class BookOrder is

*public double calcTotalPrice( )*

But if you re-write it by adding two formal parameters, it becomes:

*public double calcTotalPrice( double bookUnitPrice, int numberOfBooksOrdered )*

, which is totally wrong!!! Because as a member method of class BookOrder, ***calcTotalPrice()*** already has direct access to its private data member unitPrice and numberOfBooksOrder, and being able to access these two variables enables method ***calcTotalPrice()*** to do the job of calculating the total price, thus no other information is needed, i.e., no formal parameter is needed for the method ***calcTotalPrice()*** .

In comparison, in line 81 of class BookOrder, method ***calcSalePrice()*** is defined as:

*public double calcSalePrice(double discountRate)*

, and it needs one formal parameter discountRate, that is because the discount rate information is not defined as a private data member in class BookOrder, thus this information is supplied as a formal parameter of method ***calcSalePrice()*** . Now you should know the rule to determine when it is necessary add formal parameter(s) to a member method of a class, and when it is not necessary.

* Another way to look at the second aspect of encapsulation principle of OOP: the comment in line 78 of file BookOrder.java says that, we don’t need to use statement

double totalPrice = getUnitPrice() \* getNumberOfBooks();

to replace line 76, which is:

double totalPrice = unitPrice \* numberOfBooks;

, because method ***calcTotalPrice()*** already has direct access to the private data members: unitPrice and numberOfBooks, therefore, this no need for method ***calcTotalPrice*** ***()*** to invoke the public getters of the two private data members. Public methods have direct access to private data defined in the same class.

* A summary of the visibility of private vs public, “inside of a class” vs “outside of a class” is listed in the table below, and the red fonts that says “not visible” is exactly what the encapsulation principle stands for.

|  |  |  |
| --- | --- | --- |
| class *BookOrder* | *inside class BookOrder* | *an outside class, not the BookOrder class anymore* |
| private data members | Visible. Can be accessed directly by any method in the same class, therefore no need to call the public getter to access a private data of the same class. | Not visible. Cannot be accessed directly in any method in a different class. Thus outside a class (let’s say, class A) in which the private data is defined, we have to reply on the public getter/setter/effector of class A to read, write, and manipulate the private data of class A. |
| public method members | Visible. Can be accessed directly by any other method in the same class | Visible. Can be accessed directly in any method in an outside class. |

* This picture below also explains the encapsulation principle, and it is listed on page 13 in file “Test1StudyGUide.docx”. Page 12 of the same file has more explanations. Please cross reference these two pages in file “Test1StudyGUide.docx” with the explanations of encapsulation principle of this document. This picture also explains the table above.

The user of an encapsulated class can only access its public methods directly

Encapsulating layer:

public methods

Encapsulated content:

private data

7, follow the instructions in file “**homework4.docx**”, and work on homework 4. After you finish it, you need to submit the solution zip file to its Moodle drop box. When coding your homework, please follow all the rules in file “RulesForIndentAndAlignCode.docx”.

After unzip “UML-classDiagram.zip”, there is a file “HouseApplicationSample.java”. This is a sample file you can use to compare with your homework 4’s solution. In this sample file, I implement a house application without using an OOP class named House. But please do not confuse your homework with this sample file, and this sample file is NOT your homework 4’s solution, and you should NOT follow this sample file to work on your homework. As indicated in step 5 above, you should follow class BookOrder and TestBookOrder to work on your homework 4.

Before you work on homework 4, you can work on “bonus homework Circle class” release in this zip file **“bonusHomework-CircleClass.zip**”. To work on this bonus homework, you need to download and unzip this file, and then follow the instructions in WORD document **“bonusHomework-CircleClass.docx”**. This bonus homework is like a warming up exercise for you to code OOP class with encapsulation principle. This bonus homework is easier than homework 4. It is highly recommended to work on this bonus homework and finish it, before you work on homework 4.

8. keep working on the questions in file “Test1StudyGuide.docx”, and prepare for test 1. This file is available in Moodle folder “test 1 review lesson”.

9. use Eclipse, work on coding exercise 1.1 to 1.8 addressing the encapsulation principle of OOP from this link:

<http://www.ntu.edu.sg/home/ehchua/programming/java/J3f_OOPExercises.html#show-toc>

For this link, first work on exercise 1.1 by yourself. You need to generate the source code based on the given UML class diagram for the Circle class. Then you can compare your answer with the answer given on the website. Then you need to create an application class called TestCricle, which has the main method, and in the main method, you create two objects of Circle class using two different constructors, respectively, and then you need to call some methods of the Circle class using the Circle objects you just create, and output the results of the method callings. Again, compare your TestCircle class with the answer given on the website.

Then you can work on exercise 1.2 to 1.8, and for each exercise, you need to first create the OOP class based on the given UML class diagram of the OOP class, and then for each OOP class, you need to create an application class that has the main method, and then create several objects of the OOP class type using different constructors, respectively, and then call some method of the OOP class using the objects you just created, and output the results of the method callings, like what you did in exercise 1.1 for class TestCircle.

Another link addressing the encapsulation principle of OOP is: <http://www.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html> , and there are more explanations about the encapsulation principle of OOP in this link.

10. next lesson, we will work on Moodle folder “test 1 review lesson”, and you need to work on all the exercises in file “***Test1StudyGuide.docx”*** , in order to prepare for the test. The lesson after “test 1 review lesson” will be Moodle folder “chap 5”.