**OOP2 Problem Set 3: Aggregate classes, ‘static’, packages, and JAR archives.**

**Relates to Section 3: Units 8 – 9.**

If you cannot complete the set in the allocated lab time, be sure to spend some extra time on them

later to make up the lost ground.

1. Define the term **aggregate** (composite) **class**. If you haven’t already done so, copy the files

**Trainee.java** and **TraineeTest.java** from the x: drive to your network H: drive. Take a good

look at them, then answer the following questions. You may need to add or modify the code to

answer certain questions.

(i) What is it about the Trainee class that makes it an aggregate class?

(ii) The Trainee class is defined as a **public** class. Where can it be accessed? If the

public modifier were removed would the code still compile? If so, where can the

class be accessed now?

(iii) Notice that the Trainee constructors have no return types. Would it be possible for

them to have return types? If not, why not?

(iv) When the **startwgt** object is being created in the Trainee constructor, a Weight

constructor is called. Is this Weight constructor an explicit **no-argument**

**constructor** or a **default constructor**? Can you tell just by looking at the Trainee

class code? If not, have a look at **Weight.java** to confirm.

(v) Notice in the Trainee **multi-argument copy constructor** that there are method

calls to the “regular” mutators defined in the Trainee class itself, as well as calls to

the multi-argument Weight constructor, in order to initialize a Trainee object.

Would it have been possible, rather than using the code:

startwgt = new Weight(stones, pounds);

to have code such as

setStartWeight(stones, pounds); ?

(vi) Notice in the Trainee class’s toString() method that the toString() method

is called explicitly on the nested Weight object for display purposes. Is this call to

toString() strictly necessary here? Omit it to see the result.

(vii) The Trainee class’s multi-argument copy constructor definition takes a total of 6

arguments. Is the **order** of these arguments critical here?

(viii) A **relationship** exists within the Trainee class definition as a result of the

aggregation contained within it. What is this **type** of relationship called? Which class

is the “**owning**” class? Which is the “**nested**” class?

2. Write an **aggregate** (composite) class definition for a **House.** A house will have the

following attributes: *address*, *type*, *price* and *owner* (of type **Person**). It should have **accessor** and **mutator** methods defined for each of its attributes as well as a **toString**() method (which

only accesses the attributes **indirectly**) that displays the attributes’ state in the form indicated by

the screen capture below. It should also have a no-argument constructor which sets up the

attributes **indirectly** (via its mutator methods) to give House objects the following initial state:

[“No Address Specified”, “No Type Specified”, 0.0, “No owner specified”]. It should also have a

second constructor that takes 6 arguments in total representing the 4 attributes and should **call the**

**mutator** methods directly. You should code your class for **true encapsulation** and write a

minimalistic driver program that will test all the functionality of your class. Your code should

ensure that only **valid** price data is accepted – any negative price value should leave the price

unchanged. A possible run of your driver could be as follows:

****

3. Static constants Give the line of code that would create a **constant** static attribute called

SPEED\_OF\_LIGHT with a value of 3.0E8 (i.e. 3\*108) that would be directly accessible from

any other class [review what you know about scientific notation for numbers, and if in doubt,

consult http://en.wikipedia.org/wiki/Scientific\_notation]. Now imagine that this constant has been

defined in a public class called PhysicsStuff and it is required to use it in a separate driver

program that has access to the PhysicsStuff class. Give the line of code that would be

required in the driver to determine the energy associated with a mass of certain value based on the

formula:

energy = mass\*(speed of light)2

you can assume that energy and mass are just variables that have been declared locally in the

driver.

4. Static methods: non-instantiable class. Take a look back at your solution to problem sheet 1B

question 6 which involved creating a static method called **cube()** and calling it from the

(static) main method**.** Modify the code here by removing the reference to static in the

**cube()** method header. Try to recompile. What is the problem here? What does this tell you

about static methods? Now remove the reference to static in the **main()** method header.

Does this solve the problem? Now try to run. What does this tell you about the main() method?

5. The sample classes Account and AccountTest. Identify a static attribute and a non-static

attribute from Account. Why is one of the attributes static? Make it non-static, and recompile.

What happens? Why? Make any necessary changes so that it will compile, then try to recompile

AccountTest. What happens, and why? Change it so that it compiles.

6. What is the major difference between a *static* method and an *instance* method?

What are the major benefits associated with the creation of static attributes? Explain

briefly why all attributes of a class are not ordinarily defined to be static.

7. Including static attributes and methods in an instantiable class of your own. You will have written a class called Book, with attributes title, price, ISBN and number of pages, for Problem

set 2, Q2. Now suppose you want a class to model the books in a uniform series, all of which will

have the same price. Make a version of your Book class called SeriesBook, in which the price

attribute and all related methods are made static. Write a driver for SeriesBook which creates

some books, sets the standard book price, outputs their details, changes the standard book price

and displays it, then outputs the details of the books again.

8. Static utility methods. Write a class which contains a class (static) method called

calcOverallMark() to work out a weighted average of three marks, taking 15% of each of the first

two marks (classroom c/a and hand-in c/a) and 70% of the third mark (end-of-module exam).

The 3 marks should be supplied as arguments, and the method should return the weighted average

mark. This method should not do any input or output. In a separate class, write a driver program

which will ask the user to enter the 3 marks, calculate the result (using the method from the other

class), and output the result to a dialog. Draw a VOPC diagram to illustrate your pair of classes.

Optional Extra section, on packages and jar files, for those interested. Non-examinable.

Optional

9. Copy the programs **WeightPackage.java** and **WeightPackageTest.java** from the x: drive to

your x: drive folder and then compile the WeightPackage.java file. Notice that a folder called

**weightpackage** has now been created – why is this? What is inside this folder? Now compile

and run the driver to make sure it is “seeing” the packaged bytecode.

Optional

10. Add the import statement

**import java.io.\*;**

to the top of the WeightPackage.java file.

Now, with the **package** statement **after** the **import** statement, investigate the result. What

happens? Can you explain why this might be the case?

Optional

11. Alter the import statement in the driver to

**import weightpackage.\*;**

Although you might expect it to continue to pick up the packaged bytecode, there is now a syntax

error indicating that WeightPackage is now inaccessible. What has happened here is that there

is a **conflict in class names** that the compiler cannot resolve. The trouble is that there is aWeightPackage.java file already within the same folder as WeightPackageTest.java. There wasno problem previously because the driver referred explicitly to the WeightPackage bytecodewithin the weightpackage folder. However, when the \* is used and there is already another javafile by the same name in the same folder as the driver, the compiler will attempt to use the one inthe same folder as the driver in the first instance and of course this cannot be accessed by thedriver as it does not exist in the package weightpackage. To get over this problem, you shouldnow simply move WeightPackage.java to another folder or rename it temporarily (make sure toclose it within JCreator before moving or renaming. When you have done this, you should seethat there is now no problem using the **import weightpackage.\*;** code

Optional

12. **Delete** the **weightpackage** folder from the file system. Now **rename the package** in the

WeightPackage.java file to “**com.itt.weightpackage”** and recompile. You should now see that

there is a directory structure called **com->itt->weightpackage** and that the WeightPackage.class

bytecode resides within the weightpackage directory. Try to compile the WeightPackageTest.java

driver program now. Can you explain why it will not compile? Make the necessary **modification**

to the WeightPackageTest.java program so that it will “see” the WeightPackage class that

resides in the **weightpackage** directory.

13. Move the WeightPackageTest.java file to another directory and attempt to run it. You should

run into **problems** now. Does this make sense? If not, think about what has changed. How might

you sort this problem out? (apart from putting it back to its original location ) **Hint**: its all about

making the compiler “see” the directory structure for the package. Edit the **Configure->Options-**

**>JDK Profiles** in JCreator in order to make the environment pick up the package and its

contents. To do this you will add an extra path to the classpath for the JDK version being used,

the one for the path to **the directory containing com->itt->weightpackage.** Do this by selecting

**Edit->Add->Add Path** and **browse to the location of the required folder**. Once you have this

added make sure that you can now compile and run the driver. You will now see the **advantage**

of this solution – it means that the client (driver) program can now reside **anywhere**, rather than

strictly in the same directory as the packaged class it is referencing. This is important since

normally references in applications are to classes that reside in **external packages**.

Optional

14. R**emove the path entry you added** to JCreator in the last part, and create a folder called

“**MyClasses**” directly under C:. Now add your package structure “**com->itt->weightpackage**” to

this “**MyClasses**” directory and set up JCreator to “see” the “MyClasses” directory. Attempt to

run the WeightPackageTest.java file again. It should be fine. The beauty of the “**MyClasses”**

folder is that it will now be **automatically searched** for by the Java environment being used for

any additional classes and packages you may have added at any stage **saving you the trouble** of

manually adding entries to JCreator yourself every time you wish to reference a newly created

package structure, which can be time-consuming and error-prone. Also it helps organization as

you can now safely keep all your own user-defined packages in **a single location** on the file

system.

Optional

15. Create your own **packaged** version of the **Person.java** program that was analysed in lectures.

Call the package “**basics**” and, when compiled, add the package to the “**MyClasses**” directory.

Make the necessary modifications to the **PersonTest.java** file in order to have it “see” the

Person class that resides in the **basics** directory. When this exercise has been done, **remove the**

**basics folder from the MyClasses folder and put it directly under your C: drive**.

Optional

16. All the user-defined packages you have been creating today have been standalone and have

not been “archived” as **Java ARchive** (JAR) files. Take a look at the **classpath** for your JDK in

JCreator now (in **Configure->Options->JDK Profiles**) and you will see that it contains entries for various JAR files, most importantly the file **rt.jar** which is the **Java runtime library**

**archive** that contains every single package and bytecode within the Java API. Mine is over 36MB

in size so you can see that this archive contains heaps of executable code. Most Java pros

distribute bytecodes using jar files as it is a highly efficient approach, applet code is often

distributed to a client machine in this fashion. They are really like a special type of zipped file

and are **compressed**. It is perfectly normal to store the contents of multiple packages within a

single jar file (this is done in the rt.jar file for example).

Creating a JAR file is very straightforward. To create one from a group of packaged classes you

just issue following command from **DOS** prompt.

**jar cf MyJarFile.jar package folder structure\\*.class**

here a jar file called MyJarFile.jar will be created upon successful execution.

So, try to create your own jar file now from the packaged Person bytecode. Call the Jar file

**Trainee.jar**. Assuming that the folder basics is on you C: drive, then, in DOS, from the C:

drive issue the command

**jar cf Trainee.jar basics\\*.class**

If the prompt returns with no error messages you know that everything is okay. You can also

check that the file called Trainee.jar has indeed been created. You should now change the name

of the folder to b**asics2** to avoid any potential naming conflicts with the jar file bytecode.

The next step is simply to add the location of the newly created jar file to JCreator in order for the

IDE to “see” it. Go into **JDK Profiles**, click on your **JDK version** and then select **Edit**. Now

select **Add->Add Archive** and locate **Trainee.jar** and then select **open** to add it to the classpath.

Once it has been added, you should now open your PersonTest.java file and check to see if it

compiles and runs. It should if everything is okay with the JAR file.