Solver: CHEN BENJAMIN, HOANG VIET, WILLIS TEE TEO KIAN, YONG SHAN JIE

1)

a)

i) Inheritance is the feature to derive new classes from existing classes, allowing derived classes to have the same attributes and behavior, thus making code reusable, reducing the effort needed when implementing new classes. On top of that, derived classes may also include new capabilities. The three other main features of Object-Oriented Model are Polymorphism, Abstraction and Encapsulation.

ii) If the 2 classes that the derived class inherits from have the same method name, there would be a problem because the derived class would not know which class it should inherit this specific attribute or behavior from. This also applies if the classes to be inherited from have the same attribute name with different data types.

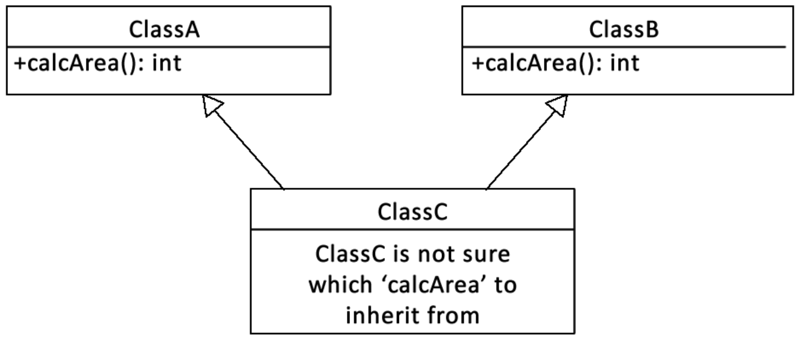

Figure 1. Problem with Multiple Inheritance

One way that Java implemented a similar concept is by using Interface, where a class may implement multiple interfaces, allowing the class to reflect the behavior of two or more 'parents'. Below is an example code:

```
public interface ClassA{
     public abstract int calcArea();
}
public interface ClassB{
     public abstract int calcArea();
}
public class ClassC implements ClassA, ClassB{
     private int base, height;
     public ClassC(){
          base = 3;
          height = 4;
     }
     public int calcArea(){
          return base*height;
```

```
        }
    }
```

b) The two benefits of polymorphism is its simplicity, and its extensibility.
**Simplicity**: A base class do not need to know the specifics of future derived class, making code easier to read and write.
**Extensibility**: When creating a class, if this new class have largely similar properties as an existing class, the new class can inherit from the existing class, and use polymorphism to only implement what is different by overriding existing class's method.

c) Constant are defined using keywords '`static final`' in java. Example:

```
public static final int MAX_HEIGHT = 180;
```

The advantages of constants are:
- When the constant value needs to be changed, it only needs to be amended once, rather than having to search for the entire code and amending lines that uses this particular value.
- Using symbolic names for constants helps the reader to understand the code better.

d)

```
public class rect{
   private int base, height;
   public rect(){
      base = 3;
      height = 4;
   }
   public void setBase(int b){
      this.base = b;
   }
}
public static void main(String args[]){
      rect r1 = new rect();
      //r1.base = 3; //INVALID
      r1.setBase(3); //Valid assignment
}
```

2)

a) **Class C:**

```
transmit(String s, int i)
transmit(String s)
transmit(int i, int j)
```

**Class D**:

```
transmit(int i, int j, int k)
transmit(String s, int i)
```

Note: ClassC does not have to implement `transmit(int i, String s)` because ClassC extends ClassA and ClassA has a concrete method `transmit(int i, String s)`

b)

i) Cannot create an object (instantiate) from abstract class, compile error in first line.

ii) Upcasting valid as ClassD is subclass of ClassB.
Abstract ClassB has an abstract method of signature (String, int), hence no compilation errors.

During runtime, we use the actual object created (ClassD), hence it will use ClassD's transmit method (which must be implemented as required by parent's abstract method of same signature).

No runtime error.

iii) Wrong family line, compile error on first line
IDE error: Type mismatch: cannot convert from ClassD to InterfaceA

iv) No explicit downcast of ClassA to ClassB.
Compile Error: Type mismatch: cannot convert from ClassA to ClassB

v) Compile time:
Line 1: Valid upcast from ClassD to ClassB (ClassD extends ClassB) -> No Error
Line 2: Reference type ClassB has method of signature (int, int) -> No Error
Line 3: Valid upcast from ClassB to ClassA (ClassB extends ClassA) -> No Error
Line 4: Reference Type ClassA has NO method of signature (String, int) -> Compilation Error

vi) Line 1: Valid upcast from ClassC to ClassA -> No Error
Line 2: Reference Type InterfaceA has method of signature (String, int) -> No Error
Line 3: InterfaceA & ClassA are of different family line -> Compilation Error (Invalid casting)

3)

a) There is no definite answer to this question, take the solution as a recommendation and with a pinch of salt.

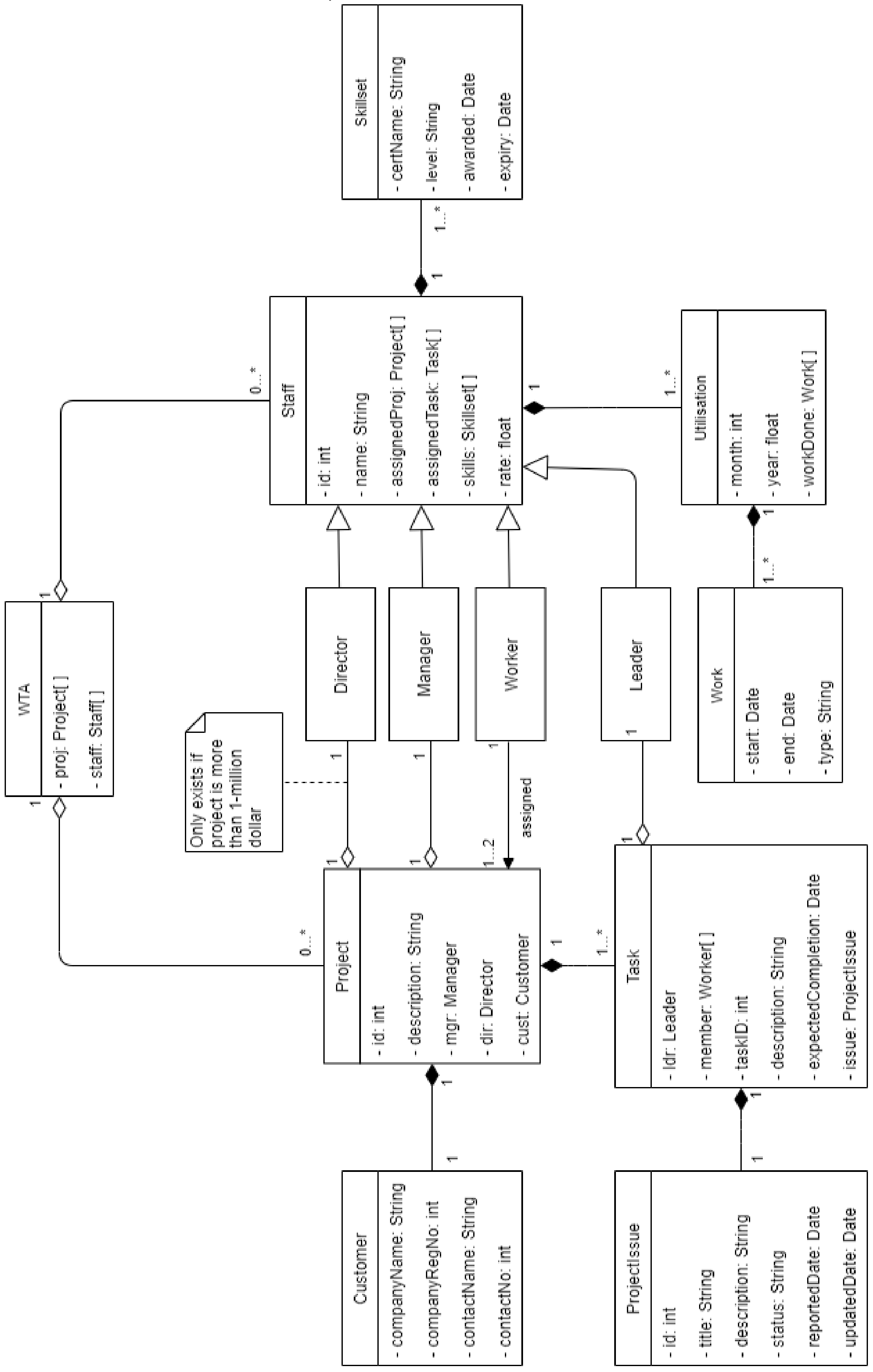

Figure 2: UML Diagram

b) Assumptions: Controller object has been created, hence methods need not be static.

```
public class Controller{
      public void accessItem(Account acc, String item_id, String action){
            AccessRight ars[] = acc.getRights();
            int result = -1;
            int i = 0;
            for (AccessRight ar : ars) {
                  /*The above for-loop statement is an Enhanced For Loop.
                  You may use conventional for-loop to perform this task.
                  But the line above simply means that for each item in
                  'ars' array, it will be referred to 'ar' for that
                  Iteration, till the last item of the array.*/

                  result = ar.verify(item_id,action);
                  if (result > -1){
                        break;
                  }
            }
            switch(result) {
              case 0:
                  readItem(item_id);
                  break;
            case 1:
                  writeItem(item_id);
                  break;
            default:
                  displayError(result);
                  break;
            }
      }
      public void readItem(String item_id){
            Viewer vw = new Viewer(item_id);
      }
      public void writeItem(String item_id){
            //Perform write item
      }
      public void displayError(int result){
            System.out.printf("Error, the result is %d", result);
      }
}
```

4)

a)

i)
```
#ifndef studinst_h
#define studinst_h
#include “studInstruct.h”
#include <string>
#include <iostream>
using namespace std;
#endif

class TeachingAsst: public Student, public Instructor{
   private:
         int minReqHours;
   public:
   TeachingAsst(int  mhours, float cgpa, float r): Student(cgpa),
         Instructor(r), minReqHours(mhours){} //CONSTRUCTOR
   ~TeachingAsst(){//destructor method}; //DESTRUCTOR
   bool checkConflict() {
               int val;
               string studCode;
               string instCode;
               for (int i = 0; i < Student::totalTutorials; i++) {
                     studCode = Student::tuts[i].getCourseCode();
                     for (int j = 0; j < Instructor::totalTutorials; j++){
                           instCode = Instructor::tuts[j].getCourseCode();
                           val = studCode.compare(instCode);
                           if (val == 0)
                                 return true;
                     }
               }
               return false;
         }
}; //REMEMBER SEMI-COLON
```

ii)
```
int main(){
   TeachingAsst* ta = new TeachingAsst(100, 5.1, 10);
   Tutorial* t1 = new Tutorial("CE2002", 10, 12);
   Tutorial* t2 = new Tutorial("CE2004", 10, 12);
   ta->registerTut(*t1);
   ta->teach(*t1);
   ta->teach(*t2);
   bool result = ta->checkConflict();
   cout << "Conflict? " << result << endl;
} //MAIN does not need a semi-colon
//checkConflict will return true.
```

b) As it is a 10m question with loads of details to cover, We don't think the examiner expect us to draw a full-fledged class diagram. Just a sketch to refer to will do.

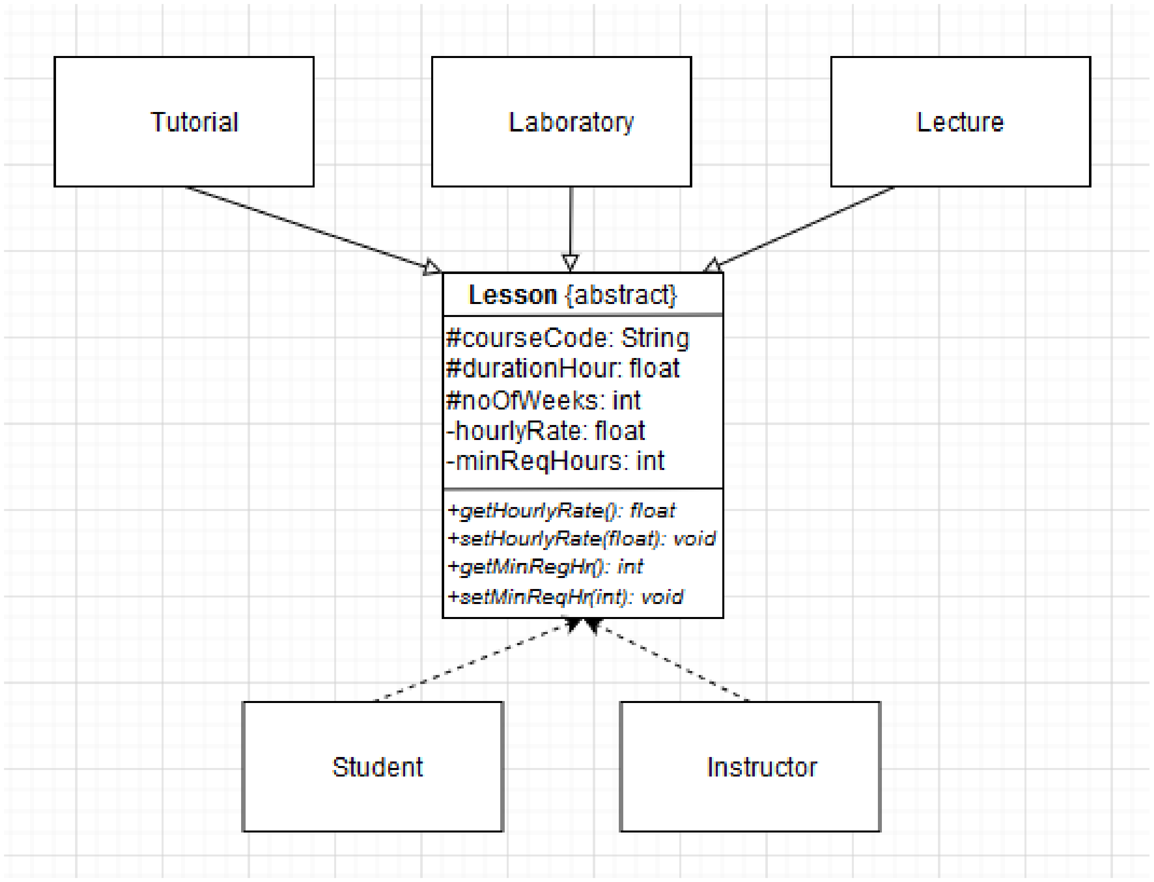

Figure 3: Enhancement to Original Code

***Enhancement 1: Lesson Abstract Class***

The three existing attributes of Tutorial (`courseCode, durationHour, noOfweeks`) are now in the new class Lesson. Tutorial, Laboratory and Lecture will extend from this abstract class.
The attribute `hourlyRate` (from Instructor) and `minReqHour` (from TeachAssist) have been moved into this class as well. They are declared as private. Note all methods in Lesson are in italics, indicating that they are all abstract methods. All classes extending from Lesson class must implement these methods.

***Enhancement 2: calculatePay() function***

Firstly, #tuts: Tutorial[5] have to be changed to #lessons: Lesson[5] inside Instructor class definition.
(Answer continues the next page)

The implementation of calculatePay() is as shown:

```
public:
     float calculatePay() {
          float totalPay = 0.0;
          for(Lesson l: Instructor::lessons) {
               totalPay += l.getHourlyRate()*l.getTotalHours();
          }
          return totalPay;
     };
```

***<u>Two SOLID Principle</u>***

For the solution, DIP (Dependency Injection Principle) and OCP (Open-Close Principle) was applied.

**OCP:**

Lesson class should be closed for modification and any new extension should be implemented in a new class that extends this Lesson class. As such we can add new features / new class to our system without having to make major modifications to existing lesson types.

**DIP:**

In the question, the high-level modules (Student and Instructor) is dependent on the low-level module (Tutorial).

Applying DIP which says:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

By giving an abstraction layer (Lesson), both modules will now rely on this Lesson Class instead

The effect of this change is that we can add new features - new class such as Laboratory and Lecture by extending Lesson class without having to modify existing code (OCP). The existing high level module class, Student and Instructor, ‘s related methods - methods that depend on Lesson’s attribute will not be affected as the new class “is-a” Lesson (DIP) (note that we subtly assume that Liskov Substitution Principle is adhered to in the new Lesson’s subclass, otherwise the contract between Student-Lesson will be broken when it comes to the new sub-class). This greatly enhances the maintainability and flexibility of our codebase.

--End of Answers--