21st CSEC – Past Year Paper Solution (2018 – 2019 Semester 1)
CE/CZ 2002 – Object-Oriented Design & Programming

1 (a) (i) Encapsulation. Access to private data can only be done through public methods (accessors). Updation of data can only be done explicitly using public methods (setters).

```
public class Person {
    private int age;
    public int getAge() {return this.age;}
    public void setAge(int newAge) {this.age = newAge;}
}
```

(ii) Only the public methods can be used by external classes. Default methods can be used by classes in the same package. The public methods exposed to external classes form the Application Programming Interface (API).

(b) `static int age;`
Use the static keyword.

Static variable	Instance variable
Can only be accessed by static methods	Can be accessed by instance and static methods
Created when the program starts, and destroyed when the program ends	Created when the object is created using the 'new' keyword and destroyed when the object is destroyed
One copy per class, shared among all objects	Multiple copies per class, one copy per object

(c) An object can include other objects as its data member. It refers to the 'has-a' relationship.

```
public void setList(ArrayList<Integer> numbers) {
    this.list = numbers;
}
```

(d) Polymorphism is the ability of an object reference to be referred as different types, as it "knows" which method to call depending on where it is in the inheritance hierarchy. For example, all Animals will walk, but the walk method is implemented differently for Cow, Duck and Snake. Hence, different walk method will be called during runtime.

The three concepts are Method Overriding, Binding, and Typecasting.

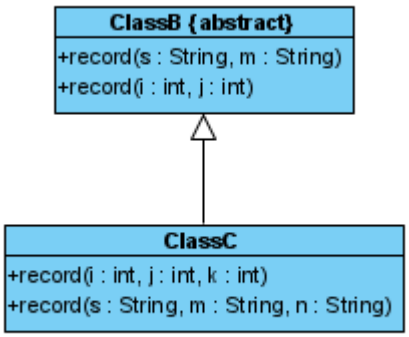
2 (a)
```
public interface InterfaceA {
      public void record(int i);
      public void record(int i, int j);
      public void record(int i, int j, int k);
}

public interface InterfaceB {
```


If there are errors, please report using the form in bit.ly/SCSEPYPError

```
        public void record(String s);
        public void record(String s, String m);
        public void record(String s, String m, String n);
}

public abstract class ClassB extends ClassA implements InterfaceA,
InterfaceB {
        public void record(String s, String m) {
                super.record(s+m);
        }

        public void record(int i, int j) {
                super.record(i+j);
        }
}
```

(b)

```
public class ClassC extends ClassB {
        public void record(int i, int j, int k) {
                super.record(i+j+k);
        }

        public void record(String s, String m, String n) {
                super.record(s+m+n);
        }
}
```



(c) **(i)** There will be compiler error, as abstract ClassB cannot be instantiated.

Line 1: Upcasting
Line 2: Upcasting (since type is still a superclass)

(ii) Line 1: Upcasting
Line 2: Downcasting, since type InterfaceB is “superclass” of ClassB
Line 3: ClassA.record(String s) is called
Line 4: ClassC.record(int I, int j, int k) is called

3 (a) (i)

```
#ifndef IFIGURE3D_H
#define IFIGURE3D_H

class IFigure3D {
      public:
            virtual double volume() = 0;
};

#endif
```

(ii)

```
#ifndef CONE_CPP
#define CONE_CPP

#include <string>
#include "Ifigure3d.h"
#include "figure2d.h"
#include "circle.h"

using std::string;
using std::cout;
using std::endl;

class Cone : public IFigure3D, public Figure2D {
      private:
            double height;
            double slanted_ht;
            Circle cir;

      public:
            Cone(string name, double ht, double s_ht, double rad) :
Figure2D(name), height(ht), slanted_ht(s_ht) {
                  cir.setRadius(rad);
            }

            double getHeight() {return height;}
            void setHeight(double h) {height = h;}

            double getSlantHt() {return slanted_ht;}
            void setSlantHt(double sh) {slanted_ht = sh;}

            double volume() {
                  double M_PI = cir.getPI();
                  return (M_PI * radius * radius * height)/3.0;
```


If there are errors, please report using the form in bit.ly/SCSEPYPError

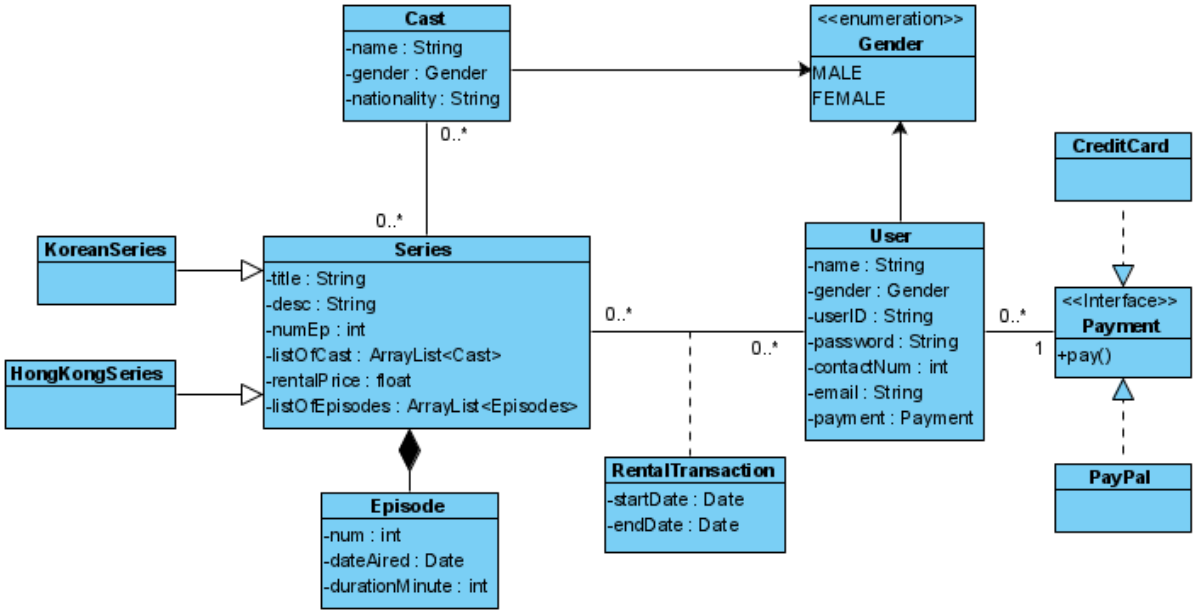
```
                }

                double area() {
                      double M_PI = cir.getPI();
                      return M_PI * radius * (slanted_ht + radius);
                }

                void print() {
                      Figure2D::print();
                      Circle::print();

                      cout << "Cone height: " << height << ", slanted
                      height: " << slanted_ht << endl;
                }
          };

          #endif
```

(b)

```
public class Dealer {
      Player p1;
      Deck deck;
      public void dealMoreCards() {
            p1.moreCard();
      }

      public int getCard() {
            int val = deck.getNextCardVal();
            return val;
      }
}

public class Player {
      Hand pHand;
      Dealer d;

      public void moreCard() {
            int total = pHand.getHandValue();
            while(total < 17) {
                  int val = d.getCard();
                  total = pHand.addToHand(val);
            }
      }
}
```


If there are errors, please report using the form in bit.ly/SCSEPYPError

4 (a)



(b) (i) First, we might want to change a class or package to add new functionalities or to improve the design. A good design will allow us to make such changes easily, without the need to access many classes. For example, in the previous example, the addition of new payment methods only involves implementing the Payment interface.

Second, we might have to change a class or package because of a change to another class or package it depends on. By segregating the entities well, we can isolate the parts that might require changes, hence allow easy improvement.

(ii) Open-Closed Principle

A module is open for extension but closed for modification. This allows us to add new functionality without the need to modify the original source code. For example, a basic Car will move, stop, and refuel. By extending from the basic Car, we can design other kinds of Cars, giving them additional features, but leaving the basic functions unchanged.

Dependency Injection Principle

High-level modules and low-level modules should depend on abstractions. By doing so, the addition of new classes will not cause a full refactoring of the codes, saving unnecessary time from software development.

Solver: Tng Jun Wei (jtng008@e.ntu.edu.sg)

If there are errors, please report using the form in bit.ly/SCSEPYPError