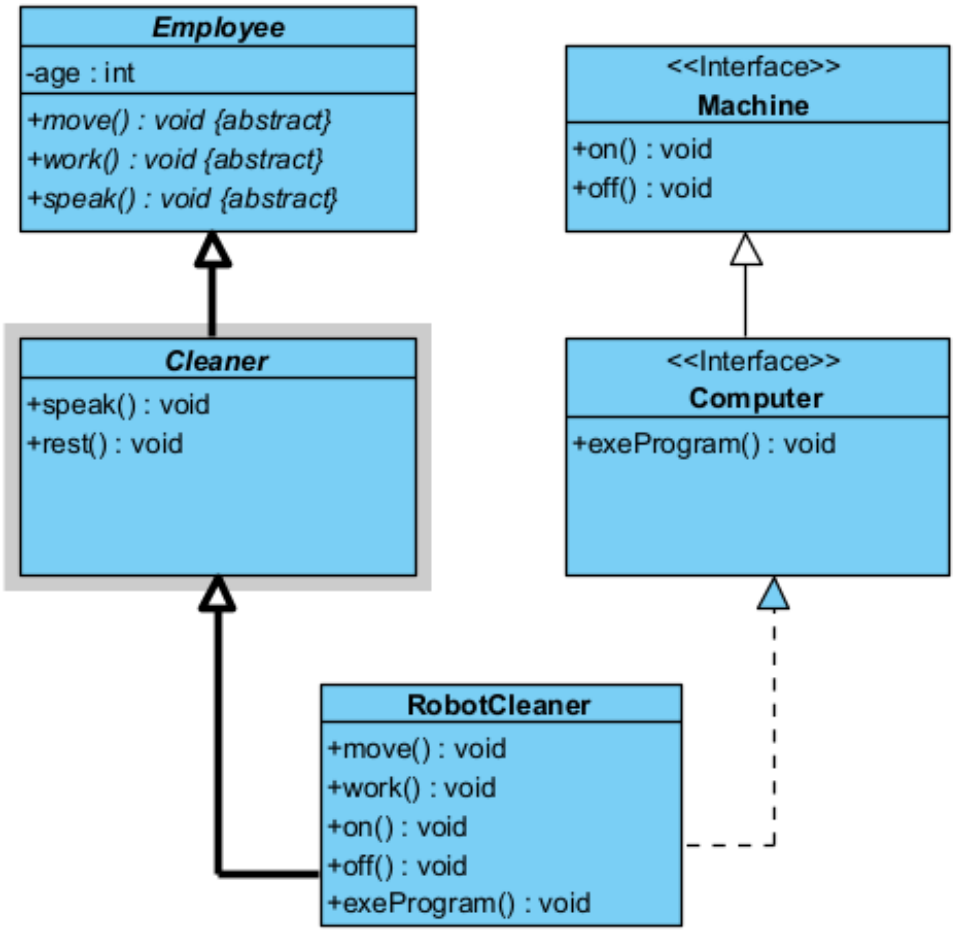
1 (a) (i)



(ii)

```
public abstract class Employee{
    private int age;
    public abstract void move();
    public abstract void work();
    public abstract void speak();
}

public interface Machine{
    public void on();
    public void off();
}

public abstract class Cleaner extends Employee{
```

```
    public void speak(){ System.out.println("Speak"); }
    public void rest(){ System.out.println("Rest"); }
}

public interface Computer extends Machine{
    public void exeProgram();
}

public class RobotCleaner extends Cleaner implements Computer{
    public void move(){ System.out.println("Move"); }
    public void work(){ System.out.println("Work"); }
    public void on(){ System.out.println("On"); }
    public void off(){ System.out.println("Off"); }
    public void exeProgram(){ System.out.println("exeProgram"); }
}
```

(b) Line 1 : B ob = new B();
Creating an object with class B. Since B is subclass of A, we will run the constructor for class A then constructor B. Output will be :
Constructor A
Constructor B

Line 2 : C oc = new C();
Same as Line 1, C is an subclass of B, which itself is a subclass of A
Constructor A
Constructor B
Constructor C

Line 3 : ob.method(100);
Variable ob itself doesn't have a function named method with an integer signature. Instead, it will trigger method(int) from the parent class
A1

Line 4 : oc.method(100);
Variable oc has a method(int) function by itself. Function method(int) from oc will run.
C1

Line 5 : ob.method(9.9);
Variable ob has a method(double) function by itself. Function method(double) from ob will run
B1
Line 6 : oc.method(9.9);
Variable oc does not have a function method(double). Continue our search for method(double) in the parent class. Class B has a function method(double). Function method(int) from b will be run
B1

2 (a) (i)

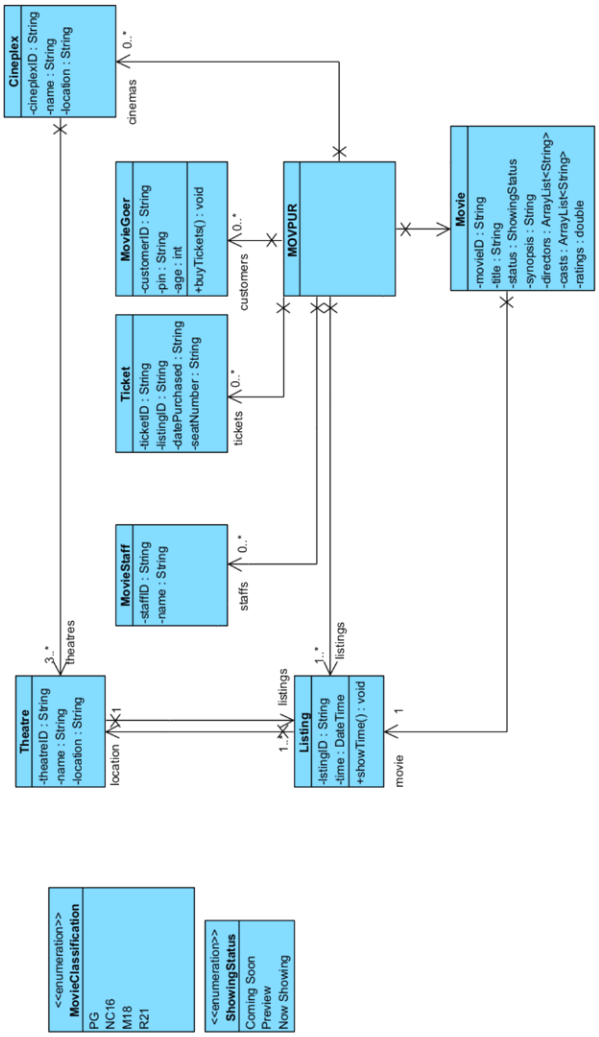
```
public class TestClass{
    private double price;
    private String name;
    static int numberOfProduct = 0;

    public TestClass(String name){
        this.price = 0.0;
        this.name = name;
        numberOfProduct++;
    }

    public static void greetings(){
        System.out.println("Hello, welcome to our product line. ");
    }

    public void update (double price, String name){
        this.price = price;
        this.name = name;
    }

    public void print(){
        System.out.printf("%s:costs $%.2f\n",this.name, this.price);
    }

    public void printNumberOfProduct(){
        System.out.printf("number : %d\n", numberOfProduct);
    }
}
```

(ii)

```
public class TestClassApp {
    public static void main(String[] args) {
        TestClass.greetings();
        TestClass myProduct = new TestClass("Amazing");
        myProduct.print();
        myProduct.update(999.99, "AmazingII");
        myProduct.print();
        myProduct.printNumberOfProduct();
        TestClass tomProduct = new TestClass("tomProduct");
        tomProduct.printNumberOfProduct();
    }
}
```

(b) Line 1, we are creating a Z object and upcasting that object into a class 'X' variable. Since X and Z both have function compare(int, int), Z's compare will override the X's compare.
Line 2, output will be 'Equal in Z'
Line 3, we are downcasting class 'X' variable into class 'Y' variable. Normally this doesn't work even with explicit downcasting. Since x1 holds the reference to class 'Z' object and Y is the parent class of Z. The downcasting is allowed.
Line 4, output will be 'Difference in Z'

3 (a)



(b)

```
import java.util.*;

public class Work{
    private String title;
    private SubWork sub;
    public Work(String title){
        this.title = title;
        this.sub = new SubWork(this);
    }

    public void init(){
```

```
        //Doing something
    }

    public void process(ArrayList<int> ar1){
        this.distribute(ar1);
    }

    public void distribute(ArrayList<int> ar1){
        ArrayList<int> ar2 = this.filter(ar1);
        sub.process(ar2);
    }

    public ArrayList<int> filter(ArrayList<int> ar1){
        //Doing something to the arrayList
        //return the modified array
        return ar1;
    }

    public String getData(){
        String data = this.loadData();
        return data;
    }

    public String loadData(){
        return "Data";
    }
}
```

4 (a) (i)

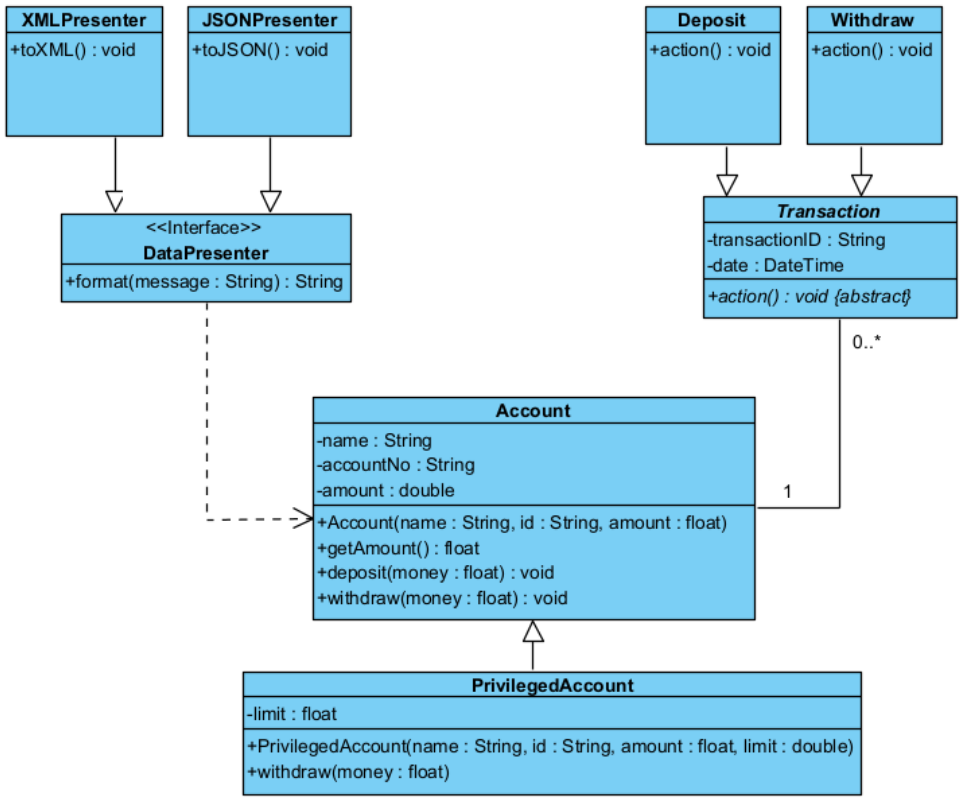
```
//account.h
#IFNDEF ACCOUNT.H
#DEFINE ACCOUNT.H

#include <iostream>

class Account{
    private:
        std::string name;
        std::string accountNo;
        float amount;
    public:
        Account(std::string name, std::string id, float amount):
name(name), accountNo(id), amount(amount){};
        ~Account(){};
        float getAmount(){ return amount;}
        void deposit(float money){ amount += money; }
```

```
        void withdraw(float money){
            if(money > amount){
                std::cout<< "Over Limit!" << std::endl;
            }
            else{
                amount -= money;
            }
        }
};
```

(ii)

```
#include <iostream>
#include "account.h"

class PrivilegedAccount: public Account{
    private:
        float limit;
    public:
        PrivilegedAccount(std::string name, std::string id, float amount,
float limit): Account(name,id,amount), limit(limit){};
        ~PrivilegedAccount(){};
        void withdraw(float money){
            if(money > amount+limit){
                std::cout<< "Over Limit!" << std::endl;
            }
            else{
                amount -= money;
            }
        }
};
```

(b) (i)



Recording of all transactions is done by generated a transaction object everytime that a transaction occurs, no matter what type of transaction

Transaction to be formatted in multiple formats is done by creating DataPresenter object that allowed account to change how the transaction would like to be presnet

(ii) **Open-Closed Principle**:

Both Transaction and DataPresenter classes are closed for modifications, but is open to extensions. This is done by allowing extension to be made but no modification of functionality. In the case for DataPresenter, its job is to format the message of transaction in a certain format. Extensions for the DataPresenter, i.e. XMLPresenter and JSONPresenter will format the message in different way.

Liskov Substitution Principle:

XMLPresenter and JSONPresenter classes can be used similarly to its parent class. Both are used for presenting transaction details to clients. Difference is both of them used different of encoding formats.

Solver: Tan Caken (C200243@e.ntu.edu.sg)