Solver: Du Youwei

Email Address: duyo0001@e.ntu.edu.sg

1. (a)

A block of memory for an object of ClassX is allocated. The address of the block of memory is stored in a memory location called object1. Object1 is a pointer points to the block of memory location.

The content of object1, which is the address of the block of memory allocated for a ClassX object, is copied and stored in another memory location called object2. There is still one block of memory allocated for ClassX object, while there are two pointers, object1 and object2, pointing to it. *(a diagram can be draw to illustrate the ideas if have enough time)*

(b)

Inheritance is an object-oriented feature that allows us to derive new classes from existing classes by absorbing their attributes and behaviors and adding new capabilities in new classes. Multiple inheritance means that a new class can be derived from two or more classes. A problem will surface when there the parent classes have methods with the same signatures but different implementations, as it is not clear which parent class should provide the method implementation for the new class. In order to avoid the ambiguity and keep everything to be simple, Java does not support multiple inheritance.

Java uses interface inheritance as an alternative. Here is an example.

```
Public interface InterfaceA {
      Void methodA ();
}
public interface InterfaceB {
      void methodA ();
}
Public interface InterfaceC extends InterfaceA, InterfaceB {
}
```

In the above example, there will be no ambiguity for InterfaceC, as methodA is not defined in InterfaceA and InterfaceB.

(c)

Final methods: final methods cannot have dynamic binding; this is because final methods cannot be overridden by subclasses.

Private methods: private methods cannot have dynamic binding, this is because private methods cannot be inherited/accessed from the parent class by the subclasses.

Static methods: static methods belong to the class. They can be inherited but do not take in the polymorphism. If we attempt to override them, they will just hide the superclass static methods instead of overriding them.

(d)

1. (a)

i.

Line	outcome	Class's method called	Type of casting	Time
1	A ClassE object is created and the address is stored in pointer a for ClassA.	ClassE	Up casting	Runtime
2	Print(x:int, y:int) is called	ClassE	-	Runtime
3	Print(s:String)	ClassB	-	Runtime
4	Create a pointer b for ClassB and stores the content of pointer a into it.	-	Up casting	Runtime
5	Print(s:String)	ClassB	-	Runtime

ii.

Line	outcome	Class's method called	Type of casting	Time
1	A ClassF object is created and the address is stored in a for InterfaceA.	ClassF	Up casting	Compile error Runtime error
2	Create a pointer c for ClassC and stores the content of pointer a into it.	-	Up casting	Runtime
3	Create a pointer g for ClassG and stores the content of pointer a into it.	-	Down casting	runtime error
4	Print(x:int)	ClassC	-	Runtime
5	Print(x:int)	ClassC	-	Runtime error

2. (a)

```
public class ClassA {
      abstract void print (String s);
      void print (int x, int y)
      {
            System.out.println (x + " " + y);
      }
}
public class ClassB extends ClassA {
      void print (int x, int y) {
            System.out.println (x+y);
      }
      void print (String s) {
            system.out.println (s);
      }
}
public class ClassD extends ClassB {
      void print (String s) {
            system.out.print (s[0]);
      }
      void print (int x, int y) {
            system.out.println (x-y);
      }
}
public class ClassE extends ClassB {
      void print (int x, int y) {
            system.out.println ( (x+y)/2);
      }
}
```

(b)

```
public class ClassApp {
      void printInt (int x, int y, ClassB b) {
            b.print(x,y); }
      void printInt (int x, int y, ClassD d) {
            d.print(x,y); }
      void printInt (int x, int y, ClassE e) {
            e.print(x,y); }

      static void main () {
            ClassB b = new ClassB ();
            printInt (5, 10, b);
            ClassD d = new ClassD ();
            printInt (5, 10, d);
            ClassE e = new ClassE ();
            printInt (5, 10, e);
      }
```

```
}
```

(c)

```
public class ClassApp {
      void printInt (int x, int y, ClassB b) {
            b.print(x,y); }

      static void main () {
            ClassB b = new ClassB ();
            printInt (5, 10, b);
            ClassD d = new ClassD ();
            printInt (5, 10, d);
            ClassE e = new ClassE ();
            printInt (5, 10, e);
      }
}
```

(d)

the output of the application class ClassApp in (b) will not be affected.

The output of the application class ClassApp in (c) will be affected. Since the print methods are all implemented as static methods, no overridden of methods and hence no polymorphism will happen. No dynamic binding occurs in ClassApp in (c). Thus, instead of calling print method in ClassB, ClassD, ClassE according to the object passed into printInt, printInt will always call the print method in ClassB.

3. (a)

```
public class IComparable {
      public:
            virtual bool operator== (IComparable &c) = 0;
}
```

(b)

```
public class SaleItem {
      private:
            String: name;
            Double: price;
      Public:
            SaleItem ();
            SaleItem (String s, double d) {
                  Name = s;
                  Price = d;
            }
            double getPrice () {
                  return price;
            }
            String getName ()
                  Return name;
```

```
            }
}
public class Sale: public IComparable {
      private:
            SaleItem: items[5];
            int: numOfItem;
      public:
            Sale() {
                  numOfItem = 0;
                  for (int i=0; i<5; 5++)
                        items[0] = NULL;
            }
            ~Sale () {
                  for (int i=0; i<numOfItem; i++)
                        delete items[i];
            }
            bool addItem (String s, double p) {
                  SaleItem temp = new SaleItem (s, p);
                  if (numOfItem < 5) {
                        items[i] = temp;
                        numOfItem ++;
                        return true;
                  }
                  return false;
            }
            double getTotal () {
                  double sum = 0;
                  for (int i=0; i<numOfItem; i++)
                        sum = sum + items[i].getPrice();
                  return sum;
            }
            bool operator== (IComparable &c) {
                 if (this.getPrice() != c.getPrice() ) return false;
                  if (this.getName() != c.getName() ) return false;
                  return true;
            }
}
```

(b)

```
class Meter {
      static void main (String args[]) {
            TestA t1 = new TestA (this);
            Bool status = false;
            Status = t1.startTest();
      }
}
class TestA {
```

```
}
```

(c)

```
public class ClassApp {
      void printInt (int x, int y, ClassB b) {
            b.print(x,y); }

      static void main () {
            ClassB b = new ClassB ();
            printInt (5, 10, b);
            ClassD d = new ClassD ();
            printInt (5, 10, d);
            ClassE e = new ClassE ();
            printInt (5, 10, e);
      }
}
```

(d)

the output of the application class ClassApp in (b) will not be affected.

The output of the application class ClassApp in (c) will be affected. Since the print methods are all implemented as static methods, no overridden of methods and hence no polymorphism will happen. No dynamic binding occurs in ClassApp in (c). Thus, instead of calling print method in ClassB, ClassD, ClassE according to the object passed into printInt, printInt will always call the print method in ClassB.

3. (a)

```
public class IComparable {
      public:
            virtual bool operator== (IComparable &c) = 0;
}
```

(b)

```
public class SaleItem {
      private:
            String: name;
            Double: price;
      Public:
            SaleItem ();
            SaleItem (String s, double d) {
                  Name = s;
                  Price = d;
            }
            double getPrice () {
                  return price;
            }
            String getName ()
                  Return name;
```

```
            }
}
public class Sale: public IComparable {
      private:
            SaleItem: items[5];
            int: numOfItem;
      public:
            Sale() {
                  numOfItem = 0;
                  for (int i=0; i<5; 5++)
                        items[0] = NULL;
            }
            ~Sale () {
                  for (int i=0; i<numOfItem; i++)
                        delete items[i];
            }
            bool addItem (String s, double p) {
                  SaleItem temp = new SaleItem (s, p);
                  if (numOfItem < 5) {
                        items[i] = temp;
                        numOfItem ++;
                        return true;
                  }
                  return false;
            }
            double getTotal () {
                  double sum = 0;
                  for (int i=0; i<numOfItem; i++)
                        sum = sum + items[i].getPrice();
                  return sum;
            }
            bool operator== (IComparable &c) {
                 if (this.getPrice() != c.getPrice() ) return false;
                  if (this.getName() != c.getName() ) return false;
                  return true;
            }
}
```

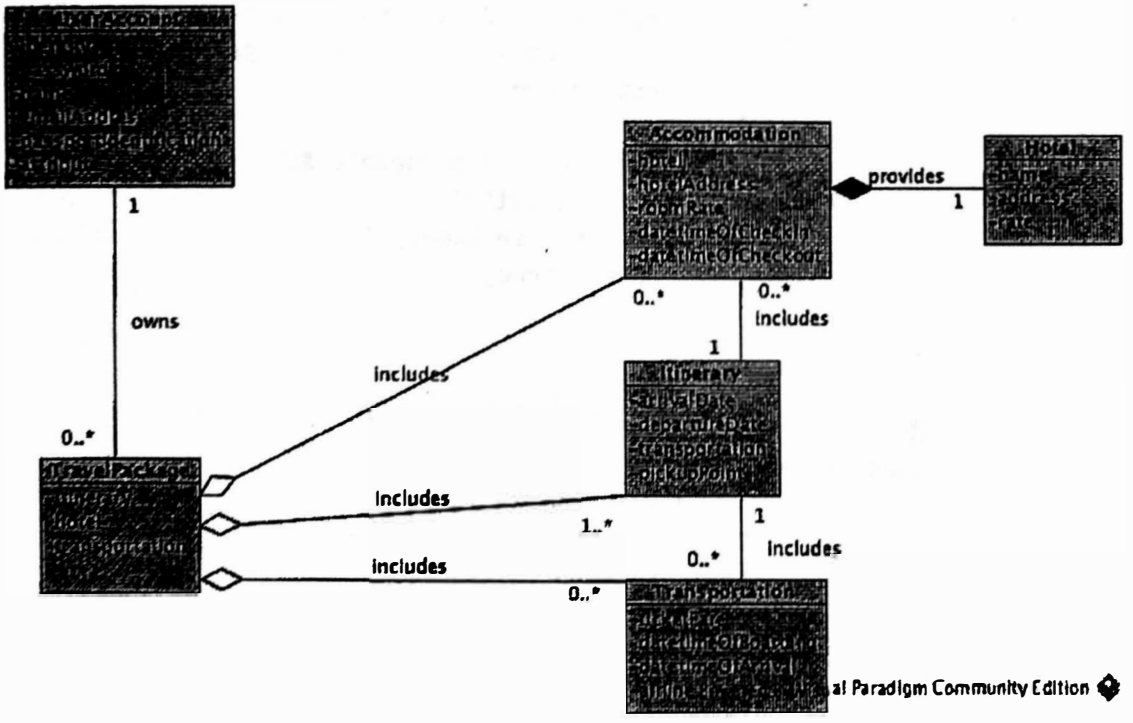
(b)

```
class Meter {
      static void main (String args[]) {
            TestA t1 = new TestA (this);
            Bool status = false;
            Status = t1.startTest();
      }
}
class TestA {
```

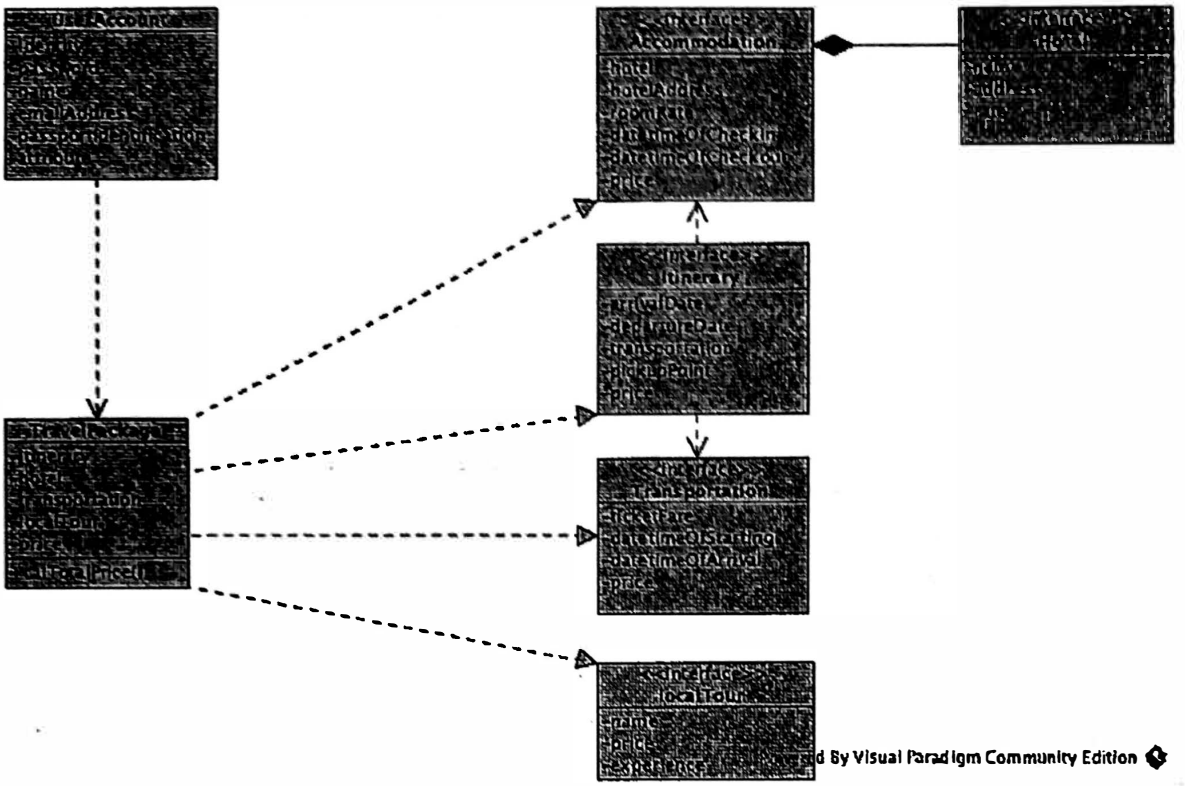
```
        Meter: mtr;
        TestA(Meter mtr) {
              Mtr = mtr;
        }
        static void main(String args[]) {
              bool status = false;
              status = start ();
        }
        bool start () {
              Double      mtr.valA = getMetricA ();
              double mtr.valB = getMetricB ();

              if ( valA * valB > 1000 ) {
                    TestB t2 = new TestB(mtr);
                    Status = T2.startTest();
              }
        }
        bool startTest () {
              return false;
        }
}
class TestB {
        startTest() {
              return false;
        }
        static void main (String srgs[])
        {}
}
```

4. (a)
(i)



(b)



To have extensibility and maintainability in the design of system, we need to follow SOLID design principles.

Single Responsibility Principle: Each class only has one responsibility, as illustrated by one role of each interface/class in the edited diagram. Hence, when a method needs to be changed, the amount of coding is reduced due to the sole responsibility of classes.

Open-Closed Principle: A module should be open for extension but closed for modification. Abstraction is the key to OCP. In this case, many interfaces are used to have loose coupling between classes.

Interface Segregation Principle: Many client specific interfaces are better than one general purpose interface. This makes the design more extensible.

In all, if a new service added into the package, a new interface can be added accordingly and Package can realize the new service. A loose link is created between Package and the new service.