20th CSEC – Past Year Paper Solution *2017-2018 Sem 1*
CE/CZ 2002 – Object Oriented Design & Programming

1) a) Polymorphism is when a program invokes a method through a superclass variable, the correct subclass version is called, based on the type of the reference stored in the superclass variable. The same method name and signature can cause different actions to occur depending on the type of object on which the method is invoked.

Benefit:
- Simplicity: code can ignore type-specific details and just interact with the base type of the family (i.e. code does not have to know if it is using the base class or any subclasses)
- Extensibility: functionality can be added by creating new classes inherited from the base class without modifying the base class and other classes derived from the base class.

b) Method overloading is using the same method name but different number of parameters or/and different parameter types.
Method overriding is when a subclass inherits properties and methods from the superclass and overrides the method implementation.

Differences:
- Method overloading is not a behavior due to inheritance but overriding is.
- Methods in overloading can have different number of parameters but not overriding.
- Methods in overriding can have different parameter types but not overriding.

c)
```
public static void methodA() {}
```

Differences:
- All objects of the same class share the same static method but for instance method, each instance of an object has its own instance method (meaning memory is allocated to instance method)
- Static methods cannot reference instance variables and methods in the class but instance methods can
- An object does not have to be instantiated to call on static method. However, for instance methods, the object has to be created first.

d) To avoid compilation error, use explicit downcasting.

```
Rectangle rect = (Rectangle) shape
```

To avoid runtime error, do check whether the object is an instance of the type being downcasted to.

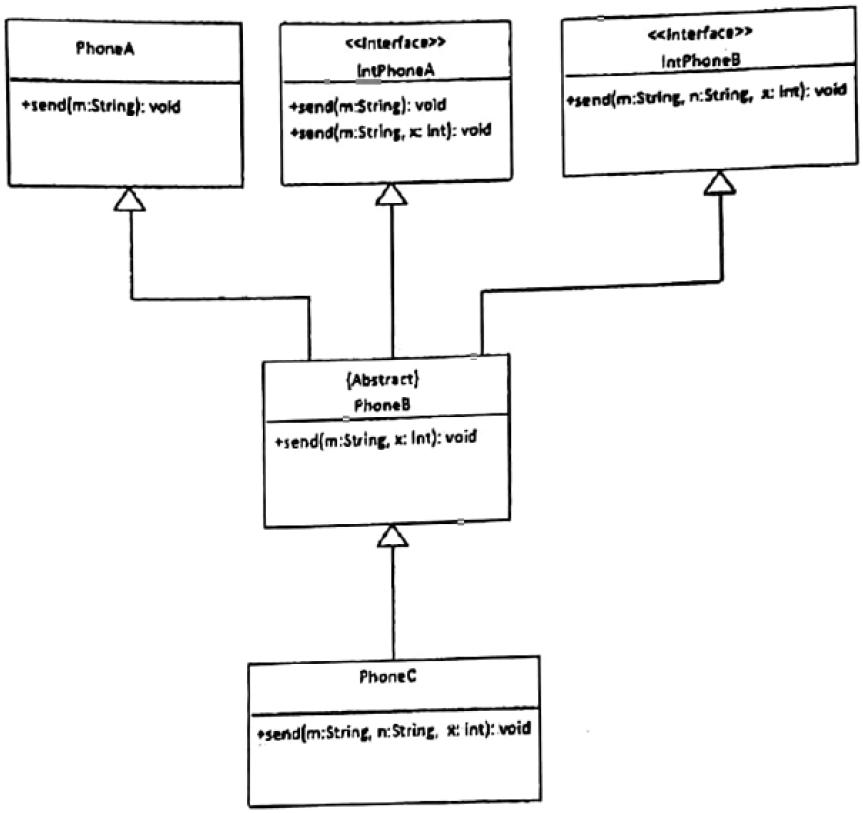
```
public boolean methodA(Object anObject) {
    if (!(anObject instanceOf Person)) return false;
    else {
        Person aPerson = (Person) anObject;
        return true;
    }
}
```

2) a)
```
public interface IntPhoneA {
    public abstract void send(String m);
    public abstract void send(String m, int x);
}
public interface IntPhoneB {
    public abstract void send(String m, String n, int x);
}
```

b)
```
public abstract PhoneB extends PhoneA implements IntPhoneA,
IntPhoneB {
    public void send(String m, int x) {
        for (int i = 0; i < x; i++) {
            super.send(m);
        }
    }
}
```

c)
```
public PhoneC extends PhoneB {
    public void send(String m, String n, int x) {
        super.send(m, x);
        super.send(n, x);
    }
}
```

d)



3) a) i)
```
using namespace std;
```

```
class Collection {
    public:
        Collection();
        ~Collection();
        virtual bool equals(string s) = 0;
        virtual void add(string s) = 0;
}
class List : public Collection {
    public:
        List();
        ~List();
        virtual string get(int index) = 0;
}
```

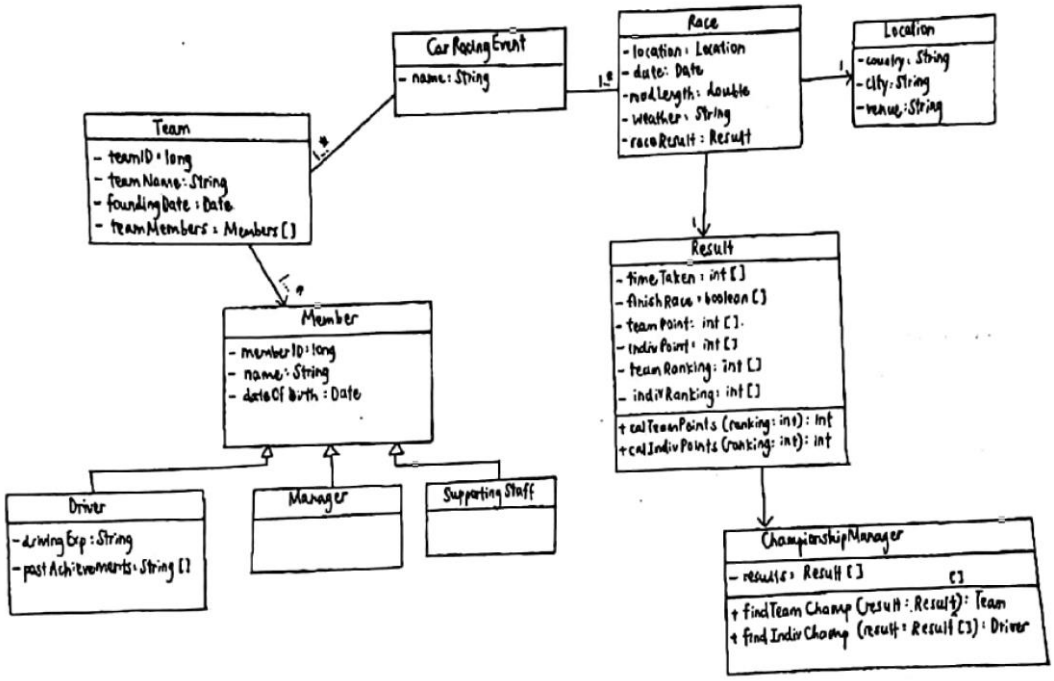
ii)
```
#include <iostream>
#include <string>
using namespace std;

class Statement {
    private:
        List words;
    public:
        Statement(int length) {words = new ArrayList(length);}
        ~Statement(){}:
        void put(string word) {words.add(word);}
}
```

b) *Editor's note:* Answer has been modified as lecturer requires code to compile.

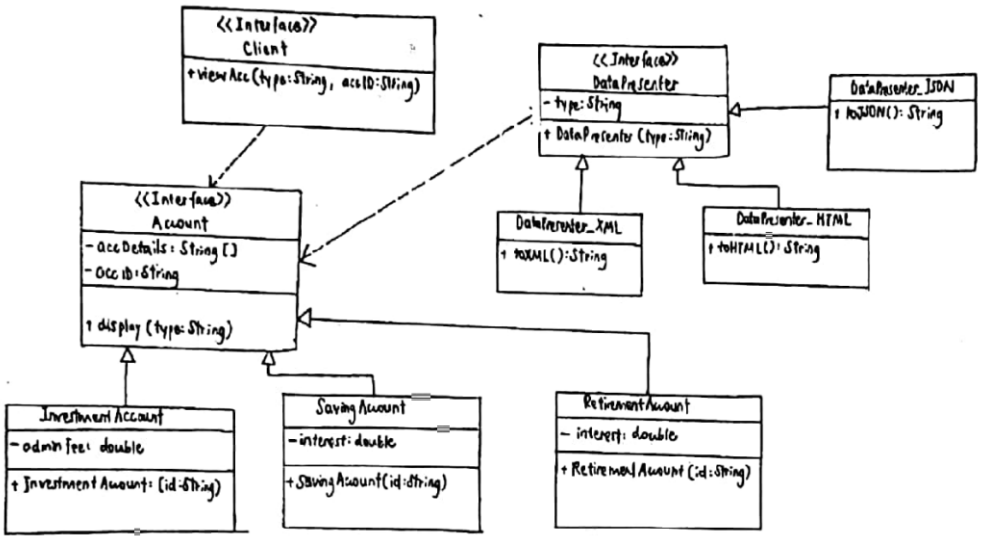
```
public Consensus {
    private SmartContract smc;
    private Consensus peer;
    public void deliverTx(ArrayList<String> tx) {processTx(tx);}
    public void processTx(ArrayList<String> tx) {
        ArrayList<String> correctTx = smc.validateTx(tx);
        if (correctTx != null) {
            orderTx(correctTx);
            peer.update(correctTx);
            ArrayList<String> verifiedTx =
                smc.verifyTxPolicy(correctTx);
            storeTx(verifiedTx);
        }
    }
    public void update(ArrayList<String> correctTx){}
    public void orderTx(ArrayList<String> correctTx){}
    public void storeTx(ArrayList<String> verifiedTx){}
}
```

4) a)



b) i) Design issue is tight coupling. The different types of accounts do not need to know how DataPresenter presents the data. The different account classes only need to focus on providing the data they have to the DataPresenter class, which will use it to decide how it wants the data to be interpreted (i.e. JSON, HTML, XML). Also, the client class has relationships to all accounts class. This means that in future cases where accounts are to be deleted or added (e.g. Retirement account class might be added in the future), the client class would have to change by adding a method to view the Retirement account class. It is unnecessary to link between client class and all the account classes.

ii)



- **Reusabiity**: The suggested design made use of Single Responsibility Principle to achieve high cohesion. Initially, DataPresenter and Account have a bidirectional relationship. However, Account does not need to know how DataPresenter presents its data. Account only needs to focus on providing the data and lend to other classes to use it and decide how they want to display the data. In the suggested design, the Account and DataPresenter assumes only one responsibility and hence if one changes, the other need not, which allows these classes to be reused for other purposes.
- **Extensibility**: Open-Closed Principle is used to allow this module to extend its functionality. Account and DataPresenter are used as interfaces so that if more Account types are added in the future (RetirementAccount) or if data should be presented in more formats in the future (DataPresenter_JSON), the interfaces are closed for modification but can be easily extendable by adding new classes to realise the respective interfaces, without having to modify the Client class.
- **Maintainability:** Interface Segregation Principle is demonstrated by using specific interfaces (Account and DataPresenter) so that the respective classes only depend on interfaces that they use. This way, if any modifications were to be made in say SavingAccount class, the interface DataPresenter and classes that realizes that interface will not be affected. Therefore, when there is a need to change, only minimum effort is required due to low coupling.

--End of Answers--

Solver: Johann Ko