

The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, A SAMPLE PAPER

OBJECT-ORIENTED METHODS

Time allowed TWO hours

Candidates must NOT start writing their answers until told to do so

Answer Question ONE and TWO others

*Marks available for sections of questions are shown in brackets
in the right-hand margin*

Only silent, self-contained calculators with a single-line display are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject specific translation dictionaries are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

DO NOT turn examination paper over until instructed to do so

1 (Compulsory Question)**NOTE:**

The final paper will have 5 multiple-choice questions, only two examples are shown...

(a) Consider the following Java classes and interface:

```
public interface IAnInterface
{
    public abstract void PrintSomething();
}

class AClass implements IAnInterface
{
    public void PrintSomething()
    {
        System.out.println("OBJ is easy!");
    }
}

class BClass implements IAnInterface
{
    public void PrintSomething()
    {
        System.out.println("OBJ is hard!");
    }
}

class CClass implements IAnInterface
{
    public void PrintSomething()
    {
        System.out.println("What is OBJ?");
    }
}
```

And the following block of code:

```
IAnInterface myInterfacePtr;
AClass a = new AClass();

myInterfacePtr = new BClass;
myInterfacePtr = a;

myInterfacePtr.PrintSomething();
```

If the above block of code was executed, using the objects defined above, what would the printed output be:

- (i) Nothing, myInterfacePtr.PrintSomething() throws an exception since the method is not defined in the interface.
- (ii) OBJ is easy!
- (iii) OBJ is hard!
- (iv) What is OBJ?
- (v) None of the above.

(4)

(f) Which of the following statements are not true about **reference counting**:

- (i) It keeps a count of the number of pointers to an object.

- (ii) it deallocates the memory for an object as soon as the object goes out of scope.
 - (iii) The user must manually work out when it is safe to release the memory (i.e. when there is no object that needs access to it)
 - (iv) Reference counting can be added to languages such as C++ which do not natively support it.
 - (v) Reference counting periodically halts the execution of the program while it works out which objects are no longer valid.
- (5)

2

- (a) Given the following class definition:

```
public class Bar
{
    private int _someIntValue;
    private String _someString;

    public Bar()
    {
        _someIntValue = 42;
        _someString = null;
    }

    public void setString(String aString) { _someString = aString; }
    public String getString() { return _someString; }

    public void setInt( int anInt) { _someIntValue = anInt; }
    public int getInt() { return _someIntValue; }
}

```

Explain how methods defined in this class are able to access the member variables in the object they are called on. (5)

- (b) The class `Foo`, below, inherits from class `Bar`. Describe, in general terms, the memory layout of this class and explain how methods inherited from `Bar` (such as `setString`) and methods defined in `Foo` are able to access the correct member variables on an object of type `Foo`.

```
public class Foo extends Bar
{
    private int _fooInt;

    public Foo()
    {
        super();
        _fooInt = 23;
    }

    public void setInt( int anInt) { _fooInt = anInt * 10; }
    public int getInt() { return _fooInt; }
}

```

(10)

- (c) Inheritance allows sub-classes to replace the implementation of certain methods. Objects of these sub-classes always use the new implementation of these methods even when they ‘pretend’ to be the super-class. Explain how object-oriented systems can use the principle of the **virtual function table** or **vtable** to ensure the correct implementation for a method is called regardless of the type of pointer to the object used. (10)

- 3 The following is a description of the card game of blackjack (also called “ponton” or “twenty one”). Using this description produce a simple object-oriented design for the system, indicating clearly the various kinds of relationship between classes and objects (e.g. aggregation, composition, inheritance etc.). [You should limit the detail you give in your design, as being appropriate for the *time available* to answer one third of this exam paper.]

The card game of blackjack is played with a pack of standard playing cards (52 cards make up a standard deck). The dealer deals a hand of two cards to each of the players including herself. Each player in turn can then ask for additional cards (known as *twisting*) until either the sum of his cards is over 21 (in which case the player is said to have *bust*), or the player decides to stop and play with the cards already obtained (i.e. *stick*). When all players have opted to stick or have bust, the banker then plays her hand, again optionally twisting

cards until she is satisfied with the hand or busts. If a player has a score closer to 21 than the banker then the player wins, otherwise the banker wins.

- (a) List all the classes required for the game and give a one sentence description of their purpose. (10)
- (b) Draw a UML diagram to illustrate the relationships between the classes in your system. The diagram should also show the major methods and variables used by the system. (If in doubt about the correct symbols used to illustrate each concept, give a key to the symbols you have used in your diagram.) (15)

4

- (a) Describe where you might use the **Abstract Factory** Pattern in an Object-Oriented program and how it works. (5)
- (b) Describe where you might use the **Factory Method** Pattern in an Object-Oriented program and how it works. (5)
- (c) The Chateaux motor group manufactures a range of cars under the Chateaux name. Each car of the new *Frio* model is produced on a totally automated production line controlled using the Java code presented below. The manufacture of each car is controlled by an object, which is created for every car made. Methods called on this object control the construction of the car. The `FrioCarProductionLine` object has an `orderCar` method which is called to order a specific car. This method is passed a `String` defining the type of car to be made (Saloon, Hatchback or Estate).

The Chateaux motor group wants to migrate its *McCoy* and *CVE* car models over to automated production lines, and so wants to adapt this code to control the new production lines, but does not want to modify the code in future for new cars. Show how the **Factory Method** pattern can be used to clean up this code and in particular add support for additional models of car. All cars implement the `ICar` interface shown and the class names for the each model and type of car are given in the following table:

Model	Type	Class Name
Frio	Saloon	<code>FrioSaloonCar</code>
	Hatchback	<code>FrioHatchbackCar</code>
	Estate	<code>FrioEstateCar</code>
McCoy	Saloon	<code>McCoySaloonCar</code>
	Hatchback	<code>McCoyHatchbackCar</code>
	Estate	<code>McCoyEstateCar</code>
CVE	Saloon	<code>CVESaloonCar</code>
	Hatchback	<code>CVEHatchbackCar</code>
	Estate	<code>CVEEstateCar</code>

Note: You do not need to provide an implementation for the car classes. Just assume they can be created as needed. (15)

```
public interface ICar
{
    void allocateMaterials();
    void buildChassis();
    void buildBody();
    void combineChassisAndBody();
    void ship();
}
```

```
public class FrioCarProductionLine
{
    public void produceCar(String type)
    {
        ICar car;

        if(type.equals("saloon"))
        {
            car = new FrioSaloonCar();
        }
        else if(type.equals("hatchback"))
        {
            car = new FrioHatchbackCar();
        }
        else if(type.equals("estate"))
        {
            car = new FrioEstateCar();
        }

        car.allocateMaterials();
        car.buildChassis();
        car.buildBody();
        car.combineChassisAndBody();
        car.ship();
    }
}
```