Rewarding Learning

ADVANCED SUBSIDIARY (AS)
General Certificate of Education
2015

Software Systems Development

Unit AS 1:

Introduction to Object Oriented Development

[A1S11]

WEDNESDAY 20 MAY, AFTERNOON

MARK SCHEME

General Marking Instructions

Introduction

Mark schemes are published to assist teachers and students in their preparation for examinations. Through the mark schemes teachers and students will be able to see what examiners are looking for in response to questions and exactly where the marks have been awarded. The publishing of the mark schemes may help to show that examiners are not concerned about finding out what a student does not know but rather with rewarding students for what they do know.

The Purpose of Mark Schemes

Examination papers are set and revised by teams of examiners and revisers appointed by the Council. The teams of examiners and revisers include experienced teachers who are familiar with the level and standards expected of students in schools and colleges.

The job of the examiners is to set the questions and the mark schemes; and the job of the revisers is to review the questions and mark schemes commenting on a large range of issues about which they must be satisfied before the question papers and mark schemes are finalised.

The questions and the mark schemes are developed in association with each other so that the issues of differentiation and positive achievement can be addressed right from the start. Mark schemes, therefore, are regarded as part of an integral process which begins with the setting of questions and ends with the marking of the examination.

The main purpose of the mark scheme is to provide a uniform basis for the marking process so that all the markers are following exactly the same instructions and making the same judgements in so far as this is possible. Before marking begins a standardising meeting is held where all the markers are briefed using the mark scheme and samples of the students' work in the form of scripts. Consideration is also given at this stage to any comments on the operational papers received from teachers and their organisations. During this meeting, and up to and including the end of the marking, there is provision for amendments to be made to the mark scheme. What is published represents this final form of the mark scheme.

It is important to recognise that in some cases there may well be other correct responses which are equally acceptable to those published: the mark scheme can only cover those responses which emerged in the examination. There may also be instances where certain judgements may have to be left to the experience of the examiner, for example, where there is no absolute correct response – all teachers will be familiar with making such judgements.

		AVAILABLE MARKS
1	Some sample words – there are others	
	base classes derived different existing	
	implements interfaces late binding methods multiple	
	private properties protected public related	
	single	
	Correct usage of words required – must be used in context	
	Sample answer	
	(i) Encapsulation means that a group of **related properties**, **methods**, and other members are treated as a **single** unit or object. Objects can control how properties are changed and methods are executed by use of the visibility modifiers **private public protected** (describe each). An object can validate values before enabling property changes. Encapsulation also makes it easier to change your implementation at a later date by letting you hide implementation details of your objects, a practice called data hiding. [1] × any 4	
	(ii) Inheritance describes the ability to create new **derived (child)** classes based on an **existing super (base)** class. The new class **inherits the visible properties and methods** and events of the **base** class, and can be customised with additional properties and methods. Base methods can be redesigned in the derived class which is known as **overriding**. **Multiple inheritance is not available** but can **implement interfaces** to overcome this. [1] × any 4	
	(iii) Polymorphism means that you can have **multiple classes derived** from a **base** class that can be **used interchangeably**, even though each class **implements** the same properties or methods in **different** ways. **Polymorphism is important to object-oriented programming because it lets you use items with the same names**, regardless of what type of object is in use at the moment through **late binding**. Allow reference to overriding methods. [1] × any 4 [12]	12
2	**(a)** Reuse of code Structured design to simplify solution Multiple developers Development time improved Simplified testing Facility of overloading and overriding Or other relevant reason [1] × any 3 [3]	

(b) sample answer:

```
public static double findCost(char style,  int size){                    [3]
      (1 mark return type, 1 mark each type of parameter)
        double cost = 0;                                                 [1]

        switch(style) {                              (correct case)      [1]
          case 'S':          cost = 120;          break;
          case 'R':          cost = 139.99;       break;
          case 'I':          cost = 215;          break;
          case 'C':          cost = 349.99;       break;
}         (1 mark any correct cost, 1 for break) or efficiency of if     [2]

if( size == 2)                                                           [1]
        cost*= 1.5;
cost*=  1.2;                                                             [1]

return cost;                                                             [1]   [10]
}
```

(c) sample answer java for style

```
do{
        p("\n\tEnter style   S – square,  R – rectangle  ….  : ");
        style = key.nextline().charAt(0);                                [1]
        if(style != 'S' && style!= 'R  ….)                               [1]
                p("\n\tError – Please re-enter");
while(style != 'S' && style!= 'R  ….))                                   [1]
```

sample answer c# for size

```
do{
      ok = true;
      try{
            Console.SetCursorPosition(5, 12);
            Console.Write("Enter size   1 – Standard,  2 – Deluxe  : }) : ");
            size = Convert.ToInt32(Console.ReadLine());
            if (size< 1 || size> 2)       // test range 1,2
                  ok = false;
            }
      catch (Exception ex) {
            ok = false;
      }
      if (!ok) {          // error message printed

            Console.SetCursorPosition(5, 25);
            Console.Write("Error – Please re-enter");
            clearErrMess();
       }
} while (!ok);
```

```
        Check integer type -     (Try- catch)                            [1]

        Entry of integer                                                 [1]
        Check range 1,2                                                  [1]
                Output           either  data entry error message        [1]
        loop                                                             [1]

cost = findCost(style, size)                                             [1]
p("\n\n\tCost of shed is : ") +cost;                                     [1]   [10]
```

AVAILABLE MARKS
23

AVAILABLE MARKS

3 (a) Constructor – C# sample

```
Loan(String loanCode, double amount, int noOfYears) {
    LoanCode =  loanCode;
    Amount = amount;

    NoOfYears = noOfYears;
}
```

[1]

[1] each x 3 [3]

GET/SET c# sample

public double Amount [2]
(I mark type, 1 mark capital letter for amount or correct accessor and name)

```
{
        get  { return amount;}
        set   { amount = value;}
}
```

[1]
[1]

OR

GET / SET java example

```
public double getAmount()
{
    return  amount;
}
    public void setAmount( double amount)   {
    this.amount= amount;
}
```

[1]
[1]
[1]
[1]

Method actualRate

public double actualRate() [2]
(1 mark return type, 1 mark name no parameters and the field 'amount' used in method)

```
{
    double rate=5.5;
```

[2]
(1 mark type/name, 1 mark initialisation)

```
    if( amount > 150000)
        rate = 6.05;
```

[1]

```
    else
        if(amount > 50000)
                rate = 5.25;
```

(1 mark correct 'if', 1 mark correct rate) [1]

```
    if( noOfYears >= 7)
        rate-= 0.5;
```

(1 mark correct 'if' OR 1 mark correct rate) [1]

```
    return rate;
}
```

[1] [16]

					AVAILABLE MARKS
(b)	(i)	**Instantiate loan object** **Loan aLoan = new Loan(loanCode, amount, noOfYears)** [3] (1 mark type, 1 mark new, 1 mark parameters)			
	(ii)	**Output** **System.out.print("\n\tActual rate of Interest : "** **+ aLoan.actualRate());** [2]	[5]		21
		(I mark write, 1 mark method call)			

4 (a) Examples

Exception
IOException
FileNotFoundException
NumberFormatException
IndexOutOfRangeException
any one or other [1]

(b)

```
try{

        //body of code – jumps to catch blocks on error
        //does not process subsequent statements
}
catch( ….  ){
        //        throw new LoanException(“   “);
        or
        //        deal with recovery

}
catch( ….    ){

        //        hierarchical blocks  ( Exception, if used, is last)

}
finally{

        //        block of code processed regardless if errors found or not

}
```

(1 mark each try/catch/finally, 1 mark for any two correct description) [5]

(c) Sample code C# public String LoanCode

```
{
    get { return loanCode; }
    set { //implement validation / throw new exception
        String str = value;
        if ( !validCode(str))
           throw new LoanException("Invalid Loan Code - Please check");
        else
           loanCode = value; }
}
```

(1 mark – structure, 2 mark throw invalid message, 1 mark setting correct value) [4]

Validation may be method or inline

AVAILABLE MARKS

```
boolean okFlag= true;

String letters = str.substring(0,2), digits = str.substring(2);
int num;

if( !letters.equals("CA") && !letters.equals("CI") && !letters.equals
("MA"))
okFlag = false;                                                    [2]
if( digits.All(Char.IsDigit))    // digits check will allow     –1111
      {
           num = int.Parse(str);
           if(num <100000 11 num>199999)
                  okFlag = false;
      }
      else
         okFlag = false;                                    [2]    [8]      14
```

or use of tryParse or try-catch or array. Contains

5 (a) abstract [1]

(b) public double calc_Income() [2]

{ (1 mark – type, 1 mark – no parameters)

return noOfRentalsToDate * ratePerDay;

} [1] [3]

(c) C#

class Car : Vehicle [1]

java

class Car extends Vehicle

fields:

private, type, name

(3 Car fields only) [1]

Constructors:

Empty constructor

Field constructor

(1 mark base/super)

(1 mark fields handled –passed to base and locallyset) [2]

Properties/GetSet [3]

(1 mark method name, 1 mark get, 1 mark set –any field)

Override toString() method or

Call of base/super.toString() [1] [8]

					AVAILABLE MARKS

(d)	**(i)**	override header method calc_Income() [2]
		Sample code:
		double increaseRate = 1, income ; [1]
		switch(satNav)
		{
		case 'B': increaseRate = 1.04; break;
		case 'C': increaseRate = 1.075; break;
		} [1]
		
		income = base.calc_income();
		return income *= increaseRate; [1]
			[5]		

	(ii)	overriding allow polymorphism	[1]		18
6	**(a)**	Vehicle [ ] vehicleArray = new Vehicle[150]; (1 mark class type, 1 mark instantiation)	[2]		
	(b)	**vehicleArray[0] = new Car(base data……car data…)** (1 mark class type, 1 mark base and derived data values)	[2]		

	(c)	**Sample answer**
		
		double totalIncome = 0; [1]
		for(int x = 0; x< vehicleArray.Length; x++) [2]
		{ (1 mark loop, 1 mark array length
		totalIncome += vehicleArray[x].calc_Income(); [3]
		(1 mark index, 1 mark method or Property, 1 mark add)
		}
		Console.WriteLine(" total Income : " + totalIncome); [2]
			[8]		12

				Total	**100**