Rewarding Learning

ADVANCED
General Certificate of Education

Software Systems Development

Unit AS 1

Introduction to Object Oriented Development

[SDV11]

Assessment

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.

Annotation

- Annotation should be completed in red ink
- Record the marks in the marks column on the right-hand side of the script
- Annotation should be clear and concise and specifically related to the mark scheme
- Marks for individual parts of questions should be clearly identified
- Marks should be totalled and circled at the top of each question

			AVAILABLE MARKS
1	F T T F T F		
	[1] each correct	[6]	6
2	**(a)** **(i)** Any **one** from: • Very inefficient if target value is located towards end of array but may improve by exiting when a higher value than target value reached • Full array searched if value not present • Reference to Big O or alternative solution	[1]	
	(ii) Binary	[1]	

	(b) public static int BinarySearch(int[] arr, int target){
	int minIndex = 0, maxIndex = arr.Length -1, midIndex;
	while (minIndex <= maxIndex){
	midIndex = (minIndex + maxIndex) / 2;
	if (target == arr [midIndex])
	return midIndex;
	else if (target < arr [midIndex])
	maxIndex = midIndex – 1;
	else
	minIndex = midIndex + 1;
	}
	return – 1;
	}		

	[1] method declaration – return type, parameters including integer and integer array [1] mid index found [1] loop accurately terminated [1] check if target < value found [1] check if target > mid value [1] adjust mid index relevant to a check [1] return index if target found (efficient i.e. **when** target found) [1] return –1 if not found	[8]	10

AVAILABLE MARKS

3 (a) (i)

```
public     Pizza ( String pizzaCode, int noRequested){
                PizzaCode =        pizzaCode;
                NoRequested =    norequested;
           }
```

[1] parameter types
[1] 2 fields initialised
[1] use of set / Property [3]

(ii) sample c# solution

```
public Boolean validPizza( String value){
    String toppings,   Int numVal;
    if(value.Length  >0&& value.Length  <6 ){
        if(value[0] == 'S' || value[0] == 'M' || value[0] == 'L' ){
            try {
                if (value. length >1)
                      toppings = value.substring  (1);
                       numVal = Convert.ToInt32(toppings);

                       char[] charArr = value.Substring(1).ToCharArray();
                       foreach (char ch in charArr)  {
                                if (ch == '0')
                       or
                       if (value.Contains('0')

                                          return false;
                                          else return true;
                 }
                 catch (FormatException e) {
                          return false;
                  }
       return true;
       }
       return false;
}
```

or alternative solution

[1] header visibility, return type
[1] declaration of variables
[1] check length of string greater than maximum 5
[1] check size as first character S –M – L
[1] check digits
[1] digits between 1 – 9
[1] correct logic for placement of return true if valid
[1] correct logic for placement of return if false [8]

(iii) sample c# solution

```
public double PizzaCost(){
      double cost = 0.0,  toppingPrice = 1.50;
      switch( pizzaCode[0] ){
      case   'S'   :   cost =  6.50;   break;
      case   'M' :   cost = 10.00;  toppingPrice*= 1.5; break;
      case   'L'   :   cost = 12.50;   toppingPrice*= 1.9; break;
            // assume pizza code is valid to exist
      }
      return  cost  +=  ( pizzaCode.Length – 1 ) * toppingPrice;
}
```

[1] header visibility, return type
[1] no parameter passed
[1] Declare variable
[1] switch / if to determine size as first character S –M – L
[1] calculate size cost
[1] calculate topping price
[1] return cost of pizza (number of toppings considered) [7]

(b) (i)

```
public string PizzaCode{
      get   {  return   pizzaCode;  }
      set   {  if ( validPizza( value )  )
                  pizzaCode = value;
            else
                  throw new PizzaException (" Pizza code must be
      size (S, M,L) followed by up to 4 digits for any requested toppings");
            }
}
```

[1] set
[1] call validation method
[1] pass parameter value
[1] check validity
[1] throw customised exception (with meaningful message) [6]
[1] assign pizza code

(ii) Use the set / Property to place the passed value in the field [1]

25

AVAILABLE MARKS

					AVAILABLE MARKS
4	(a)	(i)	pizzaArray[x]. pizzaCode	[1]	
		(ii)	pizzaArray[x]. noRequested	[1]	

(b)

```
sample c# solution
public   int      TotalNoPizzas(){
      int total =0;
      for  (int x=0;  x<  pizzaArray.Length -1;  x++){
            total += pizzaArray[x]. noRequested;
      }
      return total;
}
```

[1] header – return type and no parameter
[1] Declare variable
[1] loop
[1] running total (no mark if noRequested not used in some format)
[1] return [5]

(c)

```
sample c# solution
public   double OrderCost(){
      int orderCost =0;
      for  (int x=0;  x<  pizzaArray.Length -1;  x++){
        orderCost += pizzaArray[x]. noRequested  *  pizzaArray[x].PizzaCost();
      }
      if ( sitIn )
            orderCost  +=  TotalNoPizzas()  * 0.50;
      return orderCost;
}
```

[1] header - return type and no parameter
[1] Declare variable
[1] loop
[1] running total (no mark if noRequested not used in some format)
[1] call method PizzaCost()
[1] check sitIn, calculate and add on sitInCost to orderCost
[1] return [7]

(d) Any **three** from:
Order number/or [1]
Customer's name [1]
list of pizza codes with the number of each type ordered [1]
the total number of pizzas in the order [1] [3]

17

			AVAILABLE MARKS
5	**(a)**	Abstract classes: Any **two** from: cannot be instantiated [1] can contain abstract and non-abstract methods [1] must be extended [1] [2] Any valid alternative	
	(b) **(i)**	CalcInsurance() in the derived classes must be **implemented** [1] and **overridden** [1] [2]	
	(ii)	class TravelInsurance extends Insurance [1] correct [1]	
	(iii)	public override double CalcInsurance() [1] override [1] double **and** no parameters [2]	
	(c)	Static – Array Dynamic – List [1] mark for either [1]	8
6	**(a)**	Any **three** from: Interface fully abstract – cannot have implementation [1] No access modifiers [1] No constructors [1] A class can implement many interfaces – simulates multiple inheritance [1] Class must implement the interface methods [1] Imposes structure on implementing classes [1] [3] Any valid alternative	

	(b)	public int CompareTo(Object obj) {
		if (obj is Person) {
		Person other = obj as person;
		int result = this.surname.CompareTo(other.surname);
		if (result = 0)
		return this.firstname.CompareTo(other.firstName);
		else
		return result;
		}
		else
		throw new Exception(“Object is not a Person”);
		}
		
		[1] method declaration (visibility, type, name, parameter)
		[1] cast
		[1] CompareTo call for surname
		[1] call to forename (do not penalise incorrect CompareTo name again)
		[1] thrown exception if obj not Person
		[1] accurate alphabetic sort returned [6]
			9

7 **(a)**

```
public double virtual CalculateDeliveryCost() {
    double cost = 0.0;
    if ( weight > 49.99 ) {
        switch (  zone ){
            case 'L':  cost =1.00;  break;
            case 'N':  cost =1.50;  break;
            case 'I':  cost =2.50;  break;
        }
    }
    else if ( weight > 25.99   ) {
        switch (  zone ){
            case 'L':  cost =0.75;  break;
            case 'N':  cost =0.90;  break;
            case 'I':  cost =1.25;  break;
        }
    }
    else {
        switch (  zone ){
            case 'L':  cost =0.50;  break;
            case 'N':  cost =0.75;  break;
            case 'I':  cost =1.00;  break;
        }
    }
    return cost * weight;
}
```

[1] mark virtual or visibility
[1] return type
[1] no parameters
[1] cost initialised
[1] any correct assignment for weight
[1] any correct assignment for zone
[1] accurate return [7]

(b) C# Solution for dateDelivered initialised as DateTime.MinValue

```
public String TrackDelivery() {
    String  result;
    DateTime  currentDate =  DateTime.Now.Date;

if ( ! dateDelivered = = DateTime.MinValue)
    result = " Delivered on   "  +  dateDelivered.ToString() ;

else
    result = "Number of days since posting   "+  (currentDate – datePosted).TotalDays );
return result;
}
```

[1] header return type
[1] initialise current date
[1] check dateDelivered
[1] calculate number of days since posting
[1] return not delivered message (with number of days)
[1] return message with date parcel delivered [6]

AVAILABLE MARKS

13

			AVAILABLE MARKS

8	**(a)**	public ValuableParcel(int referenceNo, double weight, char zone,
		String description, double contentValue)
		: base(referenceNo, weight, zone)
		{
		Description = description;
		ContentValue = contentValue;
		}
		
		[1] parcel parameters
		[1] valuable parcel parameters
		[1] base call
		[1] correct base fields (disregard dateDelivered)
		[1] date NOT present
		[1] call to SET / Property of description and content value [6]	

	(b)	public override double CalculateDeliveryCost() {
		return base.CalculateDeliveryCost() + contentValue * 0.1;
		}
		
		[1] override
		[1] return type
		[1] no parameters
		[1] base call
		[1] additional insurance charge
		[1] return cost [6]
			12

		Total	**100**