TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

**EXAMINATIONS – 2017**

**TRIMESTER 1**

**COMP 112**

**INTRODUCTION TO COMPUTER SCIENCE**

**Time Allowed:** TWO HOURS ******** WITH SOLUTIONS **********

**CLOSED BOOK (SELECTED MATERIALS ONLY)**

**Permitted materials:** Printed Java Documentation is provided.

Silent non-programmable calculators or silent programmable calculators with their memories cleared are permitted.

**Instructions:** Answer all FIVE questions.

## Questions

| | **Marks** |
|---|---|
| 1. Defining a Class | [15] |
| 2. Files | [20] |
| 3. Shapes | [25] |
| 4. Interface | [30] |
| 5. Program and Assert Contract | [30] |

**Question 1. Defining a Class** **[15 marks]**

Complete the Account class on the facing page which stores information about bank accounts.

Each Account object should contain fields to store:

- **description**, containing the account description (text)
- **id**, which contains a unique six digit, zero filled, integer identifying the Account.
- **money**, which contains the money held in the account.
- **names**, which is a list of the signatories to the account.

Account should have a **constructor** that takes a String parameter containing the description and a double containing the money. These values should be stored in the relevant fields. It should also assign a unique **id**, using a static field to keep track of the next ID to assign.

Account should also have the following **methods**:

- **getNumber()** which returns the account Number.
- **getMoney()** which returns the money in the account
- **addName(String n)** which adds the name to the list of signatories.
- **toString()** method that returns a string containing all the data help in the account object.

(Question 1 continued on next page)

**(Question 1 continued)**

```
import java.util.ArrayList;
public class Account
{   private String description;
    private static int nextNumber = 1;
    private final String number;
    private double money;
    private ArrayList<String> names = new ArrayList<String>();

    public Account(String d, double w) {
        description = d;
        money = w;
        number = String.format("%06d", nextNumber++);
    }

    public String getNumber(){ return number; }

    public double getMoney(){ return money; }

    public String getDescription(){ return description;}

    public void addname(String n) { names.add(n); }

    public String toString(){
        String s = description+" " +
                   number + " "+
                   String.format("%.2f", money);
        for (String p:names) { s += (" "+p); }
        return s;
    }









}
```

**Question 2. Files** **[20 marks]**

Each line of a file stores information about individual bids in an auction. Each line consists of four integers:

**itemID:** The identification number of the item for sale,

**bidderID:** The identification number of the bidder

**value:** the value a person bid,

**reserve:** The reserve on the item

The lines of the file are sorted by itemID.

Write the auctionFile(String filename) method with a String parameter for the file name. This method must:

1. store a list of successful bidders in the successfulBidders field:
   ArrayList< *Integer* > successfulBidders= new ArrayList< *Integer* >();
2. return the total value of all items sold.

A successful bid must be greater than the reserve and must be the largest bid for the item.

**The following is an Example only**

**File contents in box:**

itemID- bidderID - value - reserve

| | | | |
|---|---|---|---|
| 1 | 10 | 111 | 20 |
| 1 | 11 | 110 | 20 |
| 1 | **12** | **112** | 20 |
| 2 | **20** | **100** | 50 |
| 3 | 30 | 50 | 100 |
| 4 | **41** | **100** | 90 |
| 4 | 42 | 70 | 90 |
| 4 | 43 | 95 | 90 |
| 5 | 51 | 10 | 10 |
| 5 | 52 | 20 | 10 |
| 5 | **53** | **30** | 10 |
| 6 | **66** | **10** | 5 |

**Store the following five elements**

12 20 41 53 66

**in successfulBidders**

**and return the value 352**

(Question 2 continued on next page)

**(Question 2 continued)**

```
public static int auctionFile(String filename)
  {
      int itemID= 0,bidderID = 0, value = 0, reserve = 0;
      int totalValue = 0;
      Scanner scan = null;
      boolean first = true;
      try{
          scan = new Scanner(new File(filename));
          boolean start = true;
          int lastID = -1, sofar = -1, sofarID = 0;
          boolean sold = false;
          while (scan.hasNext()){
              itemID = scan.nextInt ();
              bidderID = scan.nextInt ();
              value = scan.nextInt ();
              reserve = scan.nextInt ();
              if (start) {
                  lastID = itemID;
                  start = false;
              }
              if (lastID != itemID) {
                  lastID = itemID;
                  if (sold) {
                      totalValue += sofar;
                      successfulBidders.add(sofarID);
                      sold = false;
                      sofar = 0;
                  }
              }
              start = false;
              if (value <= reserve) { continue; }
              if (sofar < value){
                  sold = true;
                  sofar = value;
                  sofarID = bidderID;
              }
          }

. //You may continue your answer on the next page.
```

(Question 2 continued on next page)

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

**(Question 2 continued)**

```
            if (sold) {
                totalValue += sofar;
                successfulBidders.add(sofarID);
                sold = false;
                sofar = 0;
            }
        } catch(Exception e) {
            System.out.println(e);
        } finally { scan.close(); }
        return totalValue;
    }
.
            }
```

**Question 3. Shapes** **[25 marks]**

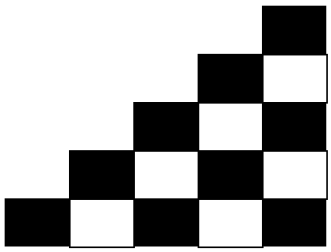
For each subquestion write a method for the Q4Shapes object to produce the designated design.

The Q4Shapes object contains the following fields.

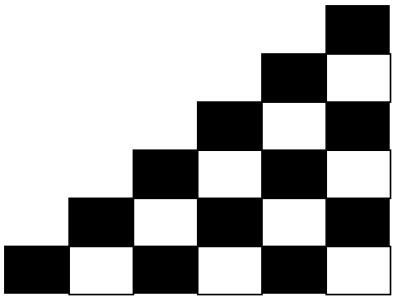
```
public class Q4Shapes
{
    private double initx = 50;      // x co-ordinate of top left corner of design
    private double inity = 100;     // y co-ordinate of top left corner of design
    private double width = 40;      // width of shape
    private double height = 30;     // height of shape
```

(a) **[10 marks]** Write the Triangle(*int* Edge) method in the Q4Shapes object. Use the fields above to **avoid hard coding** features of the design.

Calling Triangle(5) displays the image below.



Calling Triangle(6) displays the image below.



**BEWARE: the diagonal is black for both odd and even sized triangles.**

**Remember modular arithmetic.**

`Edge%2` **will be 0 for even** `Edge` **values and 1 for odd** `Edge` **values**

(Question 3 continued on next page)

**(Question 3 continued)**

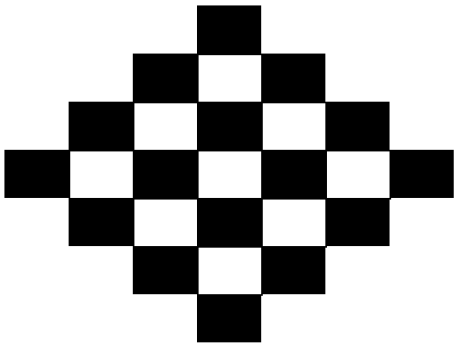
```
public void Triangle(int Edge) {



        for(int j = 0; j<Edge; j++) {
            for(int i = 0; i<Edge; i++) {
                if (j < (Edge-1) -i) continue;
                if ((i+j)%2 != Edge%2) {
                    UI. fillRect ( initx +(i*width), inity +(j*height), width, height );
                } else {
                    UI.drawRect(initx+(i*width), inity +(j*height), width, height );
                }
            }
        }
    }









}
```

(Question 3 continued on next page)

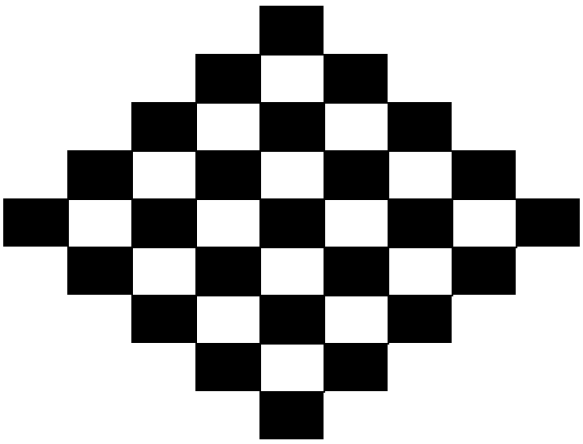
**(Question 3 continued)**

(b) **[15 marks]** Write the Diamond(*int* Wing) method in the Q4Shapes object. Use the same fields in Q4Shapes to **avoid hard coding** features of the design.

The Diamond(3) displays diamond with both height and width of $3 + 3 + 1 = 7$ elements, see image below.



Calling Diamond(4) displays a diamond with both height and width of $= 4 + 4 + 1 = 9$ elements, see image below.



(Question 3 continued on next page)

**(Question 3 continued)**

```
public void Diamond(int Wing) {

        int T = Wing*2 +1;
        for( int j = 0; j<T;j++) {
            for( int i = 0; i<T;i++) {
                if (i+j < Wing ||i+j > (2*(T-1) - Wing)) continue;
                if (i+((T-1)-j) < Wing ||i+((T-1)-j) > (2*(T-1) - Wing)) continue;
                if ((i+j)%2 == Wing%2) {
                  UI. fillRect ( initx +(i*width), inity +(j*height), width, height );
                } else {
                  UI.drawRect(initx+(i*width), inity +(j*height), width, height );
                }
            }
    }




}
```

**Question 4. Interface** **[30 marks]**

A fish farm has several locations to keep fish in. Some locations are **crowded** while other locations are not crowded.

The farmer keep several kinds of fish. Some are affected by the crowding while others are not. The following Location class has been defined.

```
public class Location {
    private int location;
    private boolean crowded;
    public Location(int l, boolean c) {
        location = l;
        crowded = c;
    }
    public boolean getCrowded(){return crowded;}
    public String toString(){
        return "loc = "+location+ " crowded = "+crowded;
    }
}
```

All weights used in the fish farm are in Kg and are always displayed to two decimal places.

The Fish interface consists of the following methods:

- grow() returns the anticipated increase of the fish's weight
- eat() returns the anticipated weight of food the fish eats
- getLocation() returns the Location object of the fish.

(a) **[5 marks]** Complete the following Fish interface:

```
public interface Fish{
        double grow();
        double eat();
        Location getLocation();




}
```

(Question 4 continued on next page)

**(Question 4 continued)**

(b) **[10 marks]** Complete the Salmon Class that (i) implements the Fish interface, and (ii) has a constructor with parameters for weight and location.

- The constructor must compute a unique tag id for the Salmon
- Salmon are not affected by crowding.
- When the salmon grows it always increases its weight by 5%
- When the salmon eats it always eats 8% of its weight in food.

```
public class Salmon          implements Fish {


        private static int nextTag = 1;
        private double weight = 0;
        private Location location;
        private String tag;
        public Salmon(double w, Location l){
            location = l;
            weight = w;
            tag = String.format("%04d", nextTag++);
        }
        public double grow(){
            return (weight * 0.05);
        }
        public double eat(){
            return (weight *0.08);
        }
        public Location getLocation(){
            return location;
        }









}
```

.

(Question 4 continued on next page)

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

**(Question 4 continued)**

(c) **[15 marks]** Complete the Trout Class that (i) implements the Fish interface, and (ii) has a constructor with parameters for weight and location.

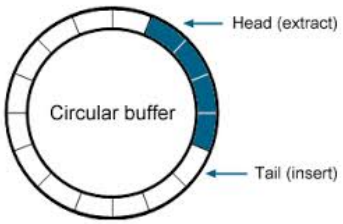
- The constructor must compute a unique tag id for the fish.
- Trout gain 7% in body weight except when crowded then they lose 2% of their body weight.
- Trout only eat when not crowded and then they eat 10% of their body weight.

```
public class Trout      implements Fish {

    private static int nextTag = 1;
    private double weight = 0;
    private Location location;
    private String tag;
    public Trout(double w, Location l){
        location = l;
        weight = w;
        tag = String.format("%04d", nextTag++);
    }
    public double grow(){
        double factor;
        if (location.getCrowded()) {
            factor = -0.02;
        } else {
            factor = 0.07;
        }
        return weight * factor;
    }
    public double eat(){
        if (location.getCrowded()) {
            return (weight * 0.0);
        } else {
            return (weight * 0.1);
        }
     }
    public Location getLocation(){
        return location;
    }


}


}
```

**Question 5. Assert, Precondition, PostCondition and Invariants 2015 [30 marks]**

The CircBufferAssert class uses an array to implement a circular buffer. Values are added to the Tail and are removed from the Head.

Both the Tail and the Head move clockwise around the circular buffer. The Tail moves when a new value is added to the buffer and the Head moves when an element is removed from the buffer.




Two private helper functions are provided:

- nextIndex(*int* x) returns the index one ahead of the index x.
  This function should be used to move the Head or Tail around the buffer.
  In a normal flat array an index i is moved by: i := i +1
  In a circular buffer an i index is moved by: i := nextIndex(i)
- bufferHolds() returns the number of elements currently in buffer

Two other helper functions need to be finished:

- empty() is true when buffer is empty, *i.e.* bufferStart and bufferHead point to the same place
- full() is true when buffer is full, *i.e.* bufferHead is one place ahead of bufferTail

All four helper functions are pure and hence can be used in both the definitions and in the assert statements.

The CircBufferAssert class has a public constructor and two public methods:

- CircBuffeAssert(*int* ss) constructs a circular buffer for ss elements
- push(*int* x) adds x to the tail of the buffer
- pull() removes an element from the head of the buffer.

Complete the class filling in the code and the assert statements.

- The numbered comments define assert statments that you need to add to each method.
- Number the assert statement as per the numbered comments.
  For example: the comment
  ** 1. all elements in buffer must be in the range -10 to +10*
  becomes
  `assert ..... : " 1. element must be between -10 and 10 ";`

The class has two invariants and to save repeated coding these checks are defined in the private method invariant that has to be called from the public methods. You need to decide where to call them from.

(Question 5 continued on next page)

**(Question 5 continued)**

(a) **[4 marks]** Complete the two methods empty() and full()

```
public class CircBufferAssert
{
    private int capacity ;
    private int bufferTail ; // index of first element if it exists
    private int bufferHead ; // index to next free entry in array
    private int [] circBuffer =null;

    //Helper functions
    /* Return the index ahead of x */
    private int nextIndex( int x){
        if (x == circBuffer.length -1) {
            return 0;}
        else {return ++x;}
    }

    /* Returns how many element are in buffer */
    private int bufferHolds() {
        if (bufferHead <= bufferTail) {
            return bufferTail - bufferHead;}
        else {
            return circBuffer .length - (bufferHead - bufferTail );}
          }

    /* Buffer is empty when bufferHead and bufferTail point to the same place */
    private boolean empty() {
                                                    return bufferHead == bufferTail;
    }
    /* The buffer is full when bufferTail is one place ahead of bufferHead */
    private boolean full (){
                                                    return bufferHead == (nextIndex( bufferTai
    }
```

(Question 5 continued on next page)

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

**(Question 5 continued)**

(b) **[14 marks]** Complete the private invariant method and the constructor.

```
/* Call invariant() at the end of the constructor and the start and end of
 *  all methods.
 * 1. all elements in buffer must be in the range −10 to +10
 * 2. buffer can not have more elements than its capacity set by the constructor
 */
private void invariant(){    //Class invariant

assert(this.bufferHolds() <= capacity) : "2 buffer exceeds maximum";
int i = bufferHead;
while ( i!= bufferTail ) {
   assert ( circBuffer[i] >= −10) : "1 element "+i+" less that -10";
   assert ( circBuffer[i] <= 10) : "1 element "+i+" greater that 10";
   i = nextIndex(i);
}


}

/**
 * 3. inputCapacity must be greater than zero
 * 4. the new buffer must be empty
 */
public CircBufferAssert(int inputCapacity)
{


   assert inputCapacity > 0 :
          "3 buffer size must be greater than zero";
     capacity = inputCapacity;
     bufferHead = 0;
      bufferTail = 0;
      circBuffer = new int[capacity+1];
      assert (empty()) : "4 initial buffer not empty";
      invariant();




}
```

(Question 5 continued on next page)

**(Question 5 continued)**

(c) **[6 marks]** Complete the pull method with pre and post conditions.

```
    /**
     * Call invariant() and
     * 5. pull must only be called if buffer contains an element
     * 6. pull must remove one element from buffer
     */
    public int pull()
     {
        assert (!empty()) : "5 pulling an empty buffer";
        invariant();
        int old = this.bufferHolds();
       bufferHead = nextIndex(bufferHead);
        assert (old -this.bufferHolds() == 1) :
                              "6 pull buffer size failure";
        invariant();
       return circBuffer[bufferHead];
.

}
```

(Question 5 continued on next page)

**(Question 5 continued)**

(d) **[6 marks]** Complete the push method with pre and post conditions.

```
    /**
     * Call invariant() and
     * 7. push must not be called if buffer is full
     * 8. pushed element i must be in range -10 to 10
     * 9. push must add one element to buffer
     */
    public void push(int i){

        assert (!full()) : "7 pushing onto a full buffer";
        invariant();
        assert (i >= -10 & i<= 10) :
            "8 pushing "+i+" element out of range -10 to 10";
        int old =this.bufferHolds();
        circBuffer[ bufferTail ] = i;
        bufferTail = nextIndex( bufferTail );
        assert (this.bufferHolds() - old == 1) :
                        "9 pushing buffer size failue";
        invariant();

.

}
```

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

* * * * * * * * * * * * * * *