SEMESTER ONE EXAMINATIONS 2007/2008
SOLUTIONS

MODULE: **Object-Oriented Programming for Engineers - EE553**

COURSE: **M.Eng./Grad. Dip./Grad. Cert. in Electronic Systems**
M.Eng./Grad. Dip./Grad. Cert. in Telecomms. Eng.
RAEC – Remote Access to Continuing Eng. Education

1(a i) Separate compilation allows us to break C++ code into separate cpp and header files. The advantage of this is that we can compile the cpp files to object code and even to libraries.On a large project this prevents us from having to compile every line of code from first principles whenever a small change is to be made.

1(a ii) Strings in Java are immutable, which means that they cannot be changed and/or grow in length. Whenever we alter a string a new string object is created that copies the contents from the previous string into the new string. This is slow, especially if a string must be altered often. A StringBuffer object can be used instead. It is mutable.

1(a iii) In the language design static methods of classes are used instead of global functions. For example Math.sqrt() allows us to call sqrt(). Static variables could also be used (and constants).

1(a iv) The Class class describes a class and is particularly useful for identifying the type of an object at runtme. Myobject.getClass() will return an object of the Class class which could be used to identify it as a java.lang.String for example.

1(a v) Java abstract classes use the abstract keyword instead of "=0" to identify an abstract method. One distinction is that a class can be abstract in Java even if it is complete. This can be used as a future design tool – to prevent objects from being created of an object so that it could be defined as abstract in the future.

1(a vi) C++ has a structure that allows us to have different types of data within the same variable. A union is particularly memory efficient as it calculates the minimum amount of memory to store the largest element in the structure. Therefore, contained data overlaps in memory.

1(a vii) Conditional operator ?. It has the form conditionalExpression ? trueExpression1 : falseExpression2, for example:

```
cout << (s.length() < t.length() ? s : t);  // Display the shorter of s and t
```

1(a) **[14 marks total – 2 for each part i to vii]**

1(b) A Constructor can be used for the task of initialising data within a class:
•It is a member function that has the same name as the class name. C++ calls the constructor of the base class and eventually calls the constructor of the derived class.
•A constructor must not have a declared return value (not even void!)
•A constructor cannot be virtual. The constructor is specific to the class that it is declared in. A new constructor must be declared for children classes as it must be specific to that class.

An example constructor code for the class Account:

```
// A constructor code example
class account{
        int myAccountNumber;
        float myBalance;
    public:
```

```
                // the constructor definition
                account(float aBalance, int anAccNumber);
};
// the constructor code implementation
account::account(int anAccNumber, float aBalance)
{
        myAccountNumber = anAccNumber;
        myBalance = aBalance;
}
// now call the constructor!
void main()
{
        account anAccount = account(35.00,34234324); //OK
        account testAccount(0.0, 34234325); //OK
        account myAccount;  //Wrong!
}
```

•If a class has a constructor then the arguments must be supplied. The constructor can be overloaded allowing other constructors to be added to the same class.
•in main() above the first two object creation calls (for anAccount and testAccount) are correct and it is up to the users style of programming. The third object construction (for myAccount) is incorrect, as the parameters are not provided to match the declared constructor!

The copy constructor is a default constructor associated with all Classes. It can be used by simply creating a new object of a class, passing the object that you wish to copy.

e.g.
Account a(1234, 10);
Account b(a);

Creates a b object of the Account class using a for that states. We can override the behaviour of this copy constructor to provide specific behaviour. For example it does not make sense to copy account numbers to new objects. Simply write a method of the form Account::Account(Account &a) and add specific behaviour.

[7 mark]

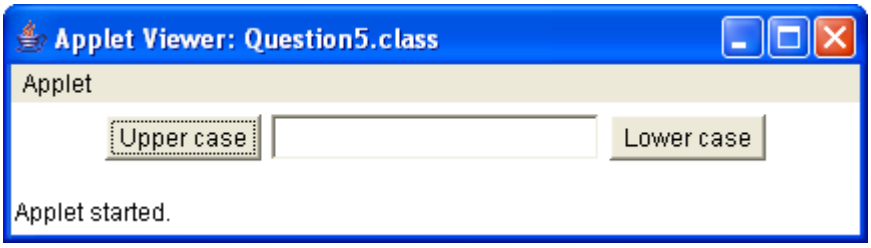
1(c)

```
public class TestParent
{
  protected int y=2; // example parent state

  public TestParent(int a) // example parent constructor
  {
    y=a;
  }
}

public class Test extends TestParent // inheritance
{
  private int x;

  public Test(int a)
  {
     super(a); // calls the parent constructor
     x = 5;
  }

  public void someMethod(int x)
```

```
  {
    this.x = x;     // this.x refers to the state of the class
    super.y = 5;        // sets the y state of the parent class to a //value of 5.

    someOtherMethod(this); // passes the current object to the
                           // someOtherMethod() method
  }
}
```

[4 marks, 1 for each this (x2), 1 for each super (x2)]

2(a)




```
// A Button Events Application

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Question5 extends Applet implements ActionListener
{

        private Button button1, button2;
        private TextField status;

        public void init()
        {
          status = new TextField(20);

          this.button1 = new Button("Upper case");
          this.button2 = new Button("Lower case");

          this.button1.addActionListener(this);
          this.button2.addActionListener(this);

          this.add(button1);
          this.add(status);
          this.add(button2);
        }

        public void actionPerformed(ActionEvent e)
        {
          if (e.getActionCommand().equals("Upper case"))
          {
            status.setText( status.getText().toUpperCase() );
          }
          else if (e.getActionCommand().equals("Lower case"))
          {
            status.setText( status.getText().toLowerCase() );
          }
```

```
        }
 }
```

HTML page:

```
<title> Question 5 Page </title>
<hr>
<applet code=Question5.class width=200 height=200>
</applet>
<hr>
```

[15 marks]
6 marks for threading, 5 marks for mouse handlers etc. 4 marks misc and understanding.

2(b)

```
public class ClockApp extends Frame implements Runnable, MouseListener {

   public ClockApp() {
         add constructor to replace init()
   }
…
   public void start() {

   }

   public void stop() {
      …
   }

   public void run() {
         …
   }

   public static void main(String args[])
         {
               ClockApp ca = new ClockApp();
               ca.start();
         }
}
```

[5 marks]
[1 change add main, 1 can const, 1 init->const, 1 call start, 1 frame and setsize, setvisible]

2(c) A Java Interface is a way of grouping methods that describe a particular behaviour. They allow developers to reuse design and they capture similarity, independent of the class hierarchy. We can share both methods and constants using Interfaces, but there can be no states in an interface. Methods in an interface are implicitly public and abstract. Constant states in an interface are implicitly public, static and final.

Interfaces differ from abstract classes in that an interface cannot have an implementation for any method, i.e. all methods are abstract. Classes can implement many interfaces, all unconstrained by a class inheritance structure.

```
 interface Demo
 {
   public static final String demoConst = "hello";
```

```
                                        // public static final can be omitted
                                        // as is implicit

  void go();            // these methods are public and abstract
  void stop();    // even if public abstract is omitted
}

class SomeClass extends Object implements Demo
{
        public void go()
        {
                // write implementation here
        }

        public void stop()
        {
                // write implementation here
        }
}

public class TestApp
{

        public static void main(String args[])
        {
                SomeClass c = new SomeClass();
                System.out.println(SomeClass.demoConst); // will print out "hello"
        }

}
```

Consider the mouse. We may write many applications that use the mouse, and within these applications, the mouse may be used in many different ways. We can define a common interface for the mouse that each one of these applications must implement. These applications then share a common defined interface, without having to create an IS-A/IS-A-PART-OF relationship.

In this example we could write an interface called StringFormatter that has one method called stringFormat. The Application to call this dialog could implement StringFormatter and implement a stringFormat method that receives the value from the Frame in 2(b) – The Frame in 2(b) must accept an object of the StringFormatter type (even though StringFormatter is an interface) – this being the case, and the fact that the Application implments the interface it can be passed as an object to the Frame.

2(d)

In Java, when you declare an array of a class, you could say:
SomeObject[] a = new SomeObject[5];
But this creates only an array of references, you must then instantiate each reference, using:
        a[1] = new SomeObject ();
Even for the default constructor (as shown). In C++, when you declare:
        SomeObject a[5];
C++ would call the default constructor for each instance of the array, and there would be no need to explicitly call 'new' again to instantiate each reference of the array.This form of initialisation only allows you to call the default constructor (or constructor with no parameters). In C++, you can create an array of objects, with each object sitting right next to each other in memory, which lets you move from one object to another, simply by knowing the address of the first object. This is impossible in Java.

In C++ you can also create arrays, of pointer or references to objects. This is more closely related to how Java arrays work. Java arrays are always an array of references to objects (except if you create arrays of the basic data types, int, float, double etc.). If you were to use an array of pointers or references, you would have an array of null references (just like Java). The syntax would be like:

```
SomeObject **f = new SomeObject *[10];
```

[5 marks]

3(a)

```
#include <iostream>
using namespace std;

class Shape
{
    protected:
      float area;
      float posX, posY;
    public:
      Shape(float, float);
      void move(float, float);
      virtual void display();
};

class Circle : public Shape
{
      float radius;
    public:
      Circle(float, float, float);
      virtual void setRadius(float);
      virtual void display();
    private:
      virtual void updateArea();
};

class Rectangle : public Shape
{
      float width, height;
    public:
      Rectangle(float, float, float, float);
      virtual void setWidth(float);
      virtual void setHeight(float);
      virtual void display();
    private:
      virtual void updateArea();
};

Shape::Shape(float x, float y): posX(x), posY(y) {}
void Shape::move(float x, float y)
{
    posX = x;
    posY = y;
}

void Shape::display()
{
    cout << "The Shape is at position (" << posX << "," << posY << ")\n";
    cout << "  has an area of " << area << endl;
}

Circle::Circle(float rad, float x, float y):
  Shape(x,y), radius(rad) { updateArea(); }
```

```
void Circle::setRadius(float rad) { radius = rad; updateArea();}
void Circle::display()
{
    Shape::display();
    cout << "  and a radius of " << radius << endl;
}
void Circle::updateArea() { area = 3.14159 * radius * radius; }

Rectangle::Rectangle(float w, float h, float x, float y):
  Shape(x,y), width(w), height(h) { updateArea();}
void Rectangle::setWidth(float w) { width = w; updateArea(); }
void Rectangle::setHeight(float h) { height = h; updateArea(); }
void Rectangle::display()
{
    Shape::display();
    cout << "  a width of " << width << " and height " << height << endl;
}
void Rectangle::updateArea() { area = width * height; }

int main()
{
  Rectangle r(100,100,20,30);
  r.display();
  Circle c(10, 20, 30);
  c.display();

  system("pause");
  return 0;
}
```

[10 marks]

~1 for each of the 8 methods, 1 for main, and 1 for correct abstract use

3 (b)

```
template<class T, int size>
class Storage
{
        T values[size];
        int next;
   public:
        T& operator [](int index)
        {
          return values[index];
        }

        bool addElement(T value)
        {
           if (next>size) return false;
           values[next++]=value;
           return true;
        }

        int getSize() { return next-1; }
};
```

[7 marks]

2 for template syntax, 2 for operator and 2 for add, 1 for size

3(c) Multiple inheritance is one aspect of C++ that leads to complications, especially when a defined class inherits from two classes.

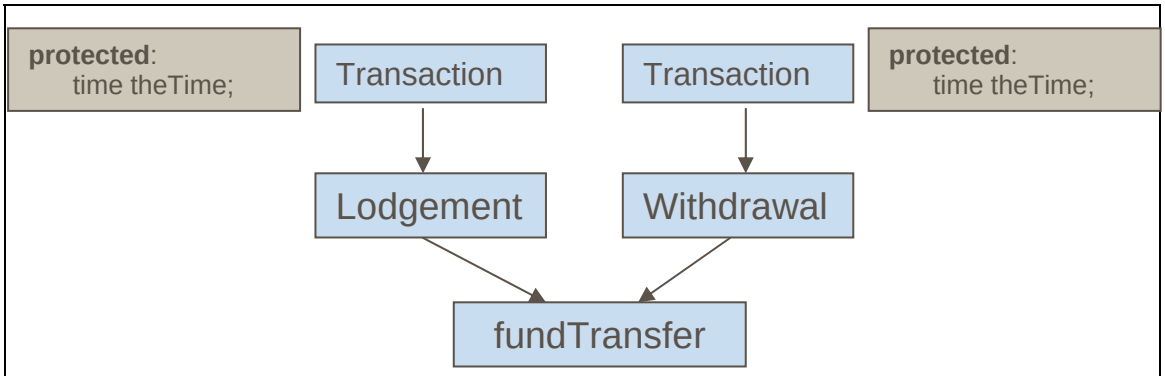


Fig Q3(c)-1 – An example where multiple base class objects are required.

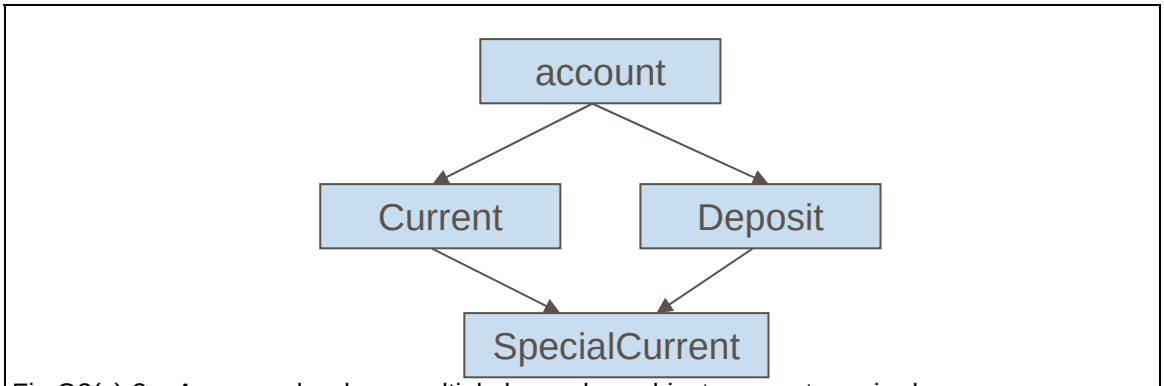


Fig Q3(c)-2 – An example where multiple base class objects are not required.

In Q3(c)-1 A Lodgement ISA Transaction, a Withdrawal ISA Transaction and at different times during execution a fundTransfer is a Lodgement or a Withdrawal.

Two separate transaction objects are being used, and they do not interfere with each other. When referring to a transaction object on the other hand, you must be careful to resolve the ambiguity that results.

```
class fundTransfer: public Lodgement,
        public Withdrawal
{       public:
        time timeTaken(){
          return abs(Lodgement::theTime - Withdrawal::theTime);
        }
};
```

Any methods that are common to both parents must be referred to it unambiguously. For example if you:

```
fundTransfer *fundPtr;
fundPtr -> display();       // not allowed – ambiguous
fundPtr -> Lodgement::display();// OK!
fundPtr -> Withdrawal::display();// OK!
```

This is awkward. It is often better to declare a new display() method in the fundTransfer class to try to avoid this disambiguation.

Virtual Functions and Multiple Inheritance

Here is an example of what goes on when using a virtual function in a multiple inheritance hierarchy. The virtual function selects the definition of that function closest to the most derived class:

In Q3(c)-2 Since account contains the balance of the account, and SpecialCurrent is a type of account, we do not want the balance to be duplicated, i.e. SpecialCurrent should only have one instance of its common indirect base class. We do this by declaring the base class to be virtual.

```
class Current: public virtual account {     // etc..
};
class Deposit: public virtual account {
        // etc..
};
```

```
class SpecialCurrent:
                public virtual Current,
                public virtual Deposit{
// etc.. This allows future reuse of SpecialCurrent
};
```

We are declaring that if either Current or Deposit should be used in a multiple inheritance hierarchy, then they should share the same instance of account with any other class that uses virtual inheritance of account. This leads to difficulties in the design process as we must determine at the time that we write deposit and current accounts that we are going to further develop a joint account. If this is not determined at this time then it will be necessary to alter and recompile these classes.

[8 marks]

4(a)

```
/*
 * Created on 20-Nov-2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
import java.awt.*;
import java.util.*;
import java.awt.event.*;
import javax.swing.*;

public class EE553Painter extends JFrame implements ActionListener{

        public class PaintCanvas extends JComponent implements MouseListener
        {
                public class JShape{
                        public int c;
                        public void draw(Graphics g) {}
                }

                public class JLine extends JShape{
                        public int x1, y1, x2, y2;
                        public int c;
                        public JLine(int x1, int y1, int x2, int y2, int cindex) {
                                this.x1=x1; this.y1=y1; this.x2 = x2; this.y2 = y2; this.c =
cindex;
                        }
                        public void draw(Graphics g){
                                g.setColor(colors[c]);
                                g.drawLine(x1, y1, x2, y2);
                        }
                }
```

```
	public class JPoint extends JShape{
		public int x1, y1;
		public int c;
		public JPoint(int x, int y, int cindex) { x1=x; y1=y; this.c=cindex; }
		public void draw(Graphics g){
			g.setColor(colors[c]);
			g.drawLine(x1,y1,x1,y1);
		}
	}

	public static final long serialVersionUID = 12346;
	private int tempX, tempY;
	private Vector<JShape> shapes= new Vector<JShape>();
	public boolean isLine = true;
	public int theColor = 0;

	public PaintCanvas()
	{
		this.addMouseListener(this);
	}

	public void paint(Graphics g)
	{
		for (int i=0; i<shapes.size(); i++)
		{
			shapes.elementAt(i).draw(g);
		}
	}

	public void mousePressed(MouseEvent e) {
		if (isLine)
		{
			tempX=e.getX();
			tempY=e.getY();
		}
		else { shapes.add(new JPoint(e.getX(), e.getY(), theColor)); }
	}
	public void mouseReleased(MouseEvent e) {
		shapes.add(new JLine(tempX, tempY, e.getX(), e.getY(), theColor));
		this.repaint();
	}
	public void mouseClicked(MouseEvent e) {}
	public void mouseEntered(MouseEvent e) {}
	public void mouseExited(MouseEvent e) {}

}

public static final long serialVersionUID = 12345;
private JButton lineB, pointB;
private PaintCanvas canvas;
private JComboBox colorCombo;
private String[] colStr = { "Black", "Blue", "Red", "Green" };
private Color[] colors = { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };

public EE553Painter() {
	super("Painter");

	canvas = new PaintCanvas();
	canvas.setPreferredSize(new Dimension(300,250));
```

```
                JPanel controls = new JPanel(new FlowLayout());
                lineB = new JButton("Draw Line");
                pointB = new JButton("Draw Point");
                colorCombo = new JComboBox();
                for(int i=0; i<colStr.length; i++)    colorCombo.addItem(colStr[i]);
                controls.add(lineB);
                controls.add(pointB);
                controls.add(colorCombo);
                colorCombo.addActionListener(this);

                this.getContentPane().setLayout(new BorderLayout());
                this.getContentPane().add(controls, BorderLayout.NORTH);
                this.getContentPane().add(canvas, BorderLayout.SOUTH);

                this.pack();
                this.setVisible(true);
        }

        public void actionPerformed(ActionEvent e)
        {
                if (e.getSource().equals(lineB)) { canvas.isLine = true;  }
                else if (e.getSource().equals(pointB)) { canvas.isLine = false;  }
                else if (e.getSource().equals(colorCombo))
                {
                        canvas.theColor = colorCombo.getSelectedIndex();
                }
        }

        public static void main(String[] args) {
                new EE553Painter();
        }

}
```

[19 marks]
Sample marks: 4 for interface, 4 for canvas, 4 for good shape structure, 4 for event structure, 4 for store and correct paint]

4(b) C++ introduces new explicit casts that identify the rationale behind the cast, clearly identifies that a cast is taking place and confirms type-safe conversions. These casts are:

- **static_cast** for well-behaved casts such as automatic conversions, narrowing conversions (e.g. a float to int), a conversion from void*, and safe downcasts (moving down the inheritance hierarchy) for non-polymorphic classes. If the base class is a virtual base class then you must use a dynamic_cast
- **dynamic_cast** is used for type-safe downcasting. The format of a dynamic cast is dynamic_cast <type>(expression) and requires that the expression is a pointer if the type is a pointer type. If a dynamic_cast fails then a null pointer is returned. The dynamic_cast is actually safer than the static_cast as it performs a run-time check, to check for ambigious casts (in the case of multiple-inheritance).
- **const_cast** is used for casting away const or volatile.
- **reinterpret_cast** is the most dangerous cast. It allows us to convert an object into any other object for the purpose of modifying that object - often for low-level "bit-twiddling" as it is known.

[6 marks total]

5(a)

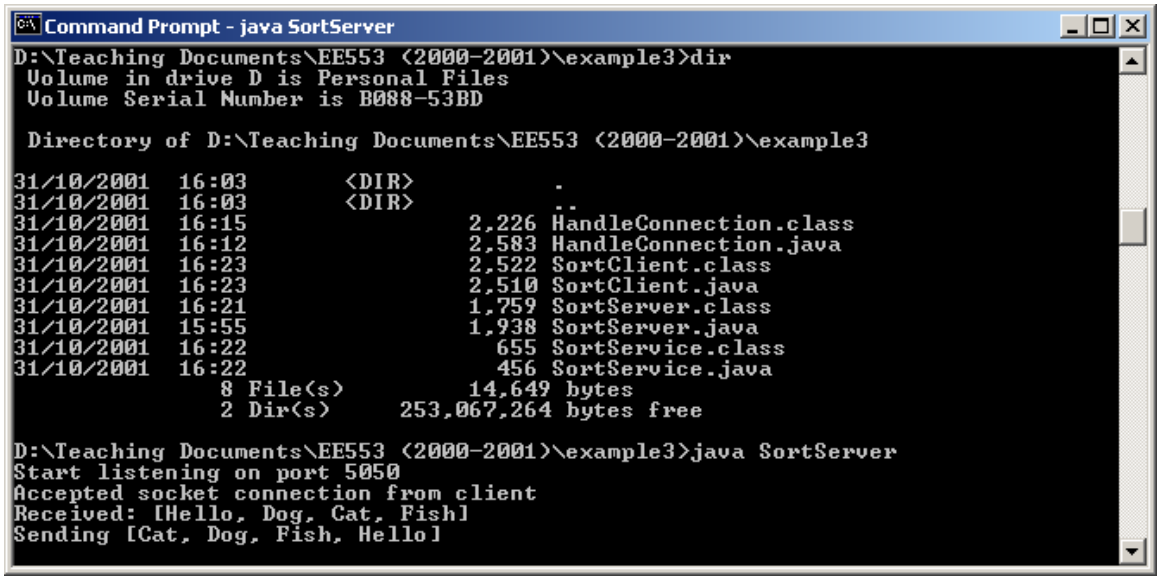
```
#include<string>
#include<iostream>

using namespace std;

class Account
{
	int actNum;
	float balance;
public:
	Account(int, float);
	virtual void display();
	void nvDisplay();
};

class Deposit: public Account
{
	float iRate;
public:
	Deposit(int, float, float);
	virtual void display();
	void nvDisplay();
};

Account::Account(int num, float theBalance):
	actNum(num), balance(theBalance) {}

void Account::display()
{
	cout << "The account number is: " << actNum << endl;
	cout << "The balance is: " << balance << endl;
}

void Account::nvDisplay()
{
	cout << "The account number is: " << actNum << endl;
	cout << "The balance is: " << balance << endl;
}

Deposit::Deposit(int num, float theBalance, float rate):
	Account(num, theBalance), iRate(rate) {}

void Deposit::display()
{
	Account::display();
	cout << "The Interest Rate is: " << iRate << endl;
}

void Deposit::nvDisplay()
{
	Account::nvDisplay();
	cout << "The Interest Rate is: " << iRate << endl;
```

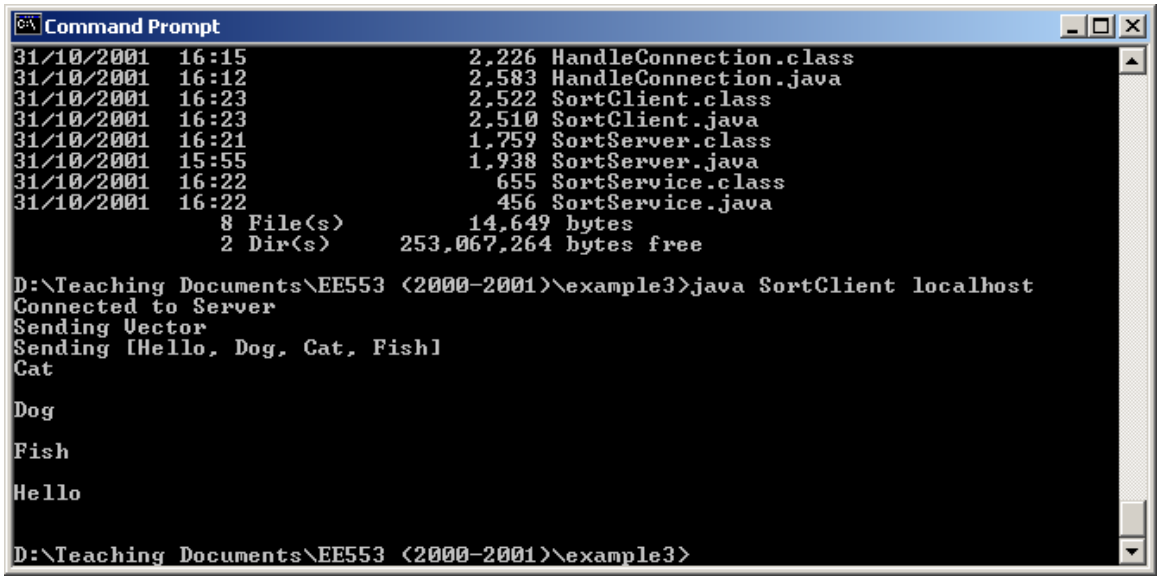
```
}

int main(void)
{
        Account *ptr = new Deposit(12345, 100.00f, 0.05f);
        ptr->display();
        ptr->nvDisplay();
}
```

[9 marks total]
[5 marks for virtual example]
[9 marks for non-virtual example]
INDICATIVE - SUBJECT TO SOLUTION PRESENTED.

5(c)

Solution:





```
// The Sort Client - written by Derek Molloy

import java.net.*;
import java.io.*;
import java.util.*;

public class SortClient
{

   private Socket socket = null;
   private ObjectOutputStream os = null;
   private ObjectInputStream is = null;

     // the constructor expects the IP address of the server - the port is fixed
   public SortClient(String serverIP)
   {
              if (!connectToServer(serverIP))
     {
                        System.out.println("Cannot open socket connection...");
              }
    }

     private boolean connectToServer(String serverIP)
   {
              try                  // open a new socket to port: 5050
     {
                        this.socket = new Socket(serverIP,5050);
                        this.os = new ObjectOutputStream(this.socket.getOutputStream());
                        this.is = new ObjectInputStream(this.socket.getInputStream());
                        System.out.print("Connected to Server\n");
        }
      catch (Exception ex)
      {
                        System.out.print("Failed to Connect to Server\n" + ex.toString());
                        System.out.println(ex.toString());
                        return false;
        }
              return true;
    }

     private void sortVector(Vector v)
   {

              System.out.println("Sending Vector");
              this.send(v);
              Vector theReturn = (Vector)receive();
              if (theReturn != null)
     {
                        for (int i=0; i<v.size(); i++)
                        {
                                  String s = (String) theReturn.elementAt(i);
                                  System.out.println(s+"\n");
                        }

      }
    }

     // method to send a generic object.
     private void send(Object o) {
              try
```

```
        {
                        System.out.println("Sending " + o);
                        os.writeObject(o);
                        os.flush();
                }
        catch (Exception ex)
        {
                        System.out.println(ex.toString());
                }
      }

      // method to receive a generic object.
      private Object receive()
    {
                Object o = null;
                try
        {
                        o = is.readObject();
                }
        catch (Exception ex)
        {
                        System.out.println(ex.toString());
                }
                return o;
      }

      static public void main(String args[])
    {
                Vector testVector = new Vector();
                testVector.add("Hello");
                testVector.add("Dog");
                testVector.add("Cat");
                testVector.add("Fish");


                if(args.length>0)
                {
                        SortClient theApp = new SortClient(args[0]);
                        try
           {
              theApp.sortVector(testVector);
                        }
           catch (Exception ex)
           {
                                System.out.println(ex.toString());
                        }
                }
                else
                {
                        System.out.println("Error: you must provide the IP of the server");
                        System.exit(1);
                }
                System.exit(0);
      }
}

// The DateServer - by Derek Molloy

import java.net.*;
import java.io.*;
```

```
public class SortServer
{
   public static void main(String args[])
   {
      ServerSocket serverSocket = null;
      try
      {
         serverSocket = new ServerSocket(5050);
         System.out.println("Start listening on port 5050");
      }
      catch (IOException e)
      {
         System.out.println("Cannot listen on port: " + 5050 + ", " + e);
         System.exit(1);
      }
      while (true) // infinite loop - wait for a client request
      {
         Socket clientSocket = null;
         try
         {
            clientSocket = serverSocket.accept();
            System.out.println("Accepted socket connection from client");
         }
         catch (IOException e)
         {
            System.out.println("Accept failed: 5050 " + e);
            break;
         }     // create a new thread for the client
         HandleConnection con = new HandleConnection(clientSocket);
         if (con == null)
         {
            try
            {
               ObjectOutputStream os = new
     ObjectOutputStream(clientSocket.getOutputStream());
               os.writeObject("error: Cannot open socket thread");
               os.flush();
               os.close();
            }
            catch (Exception ex)
            {
               System.out.println("Cannot send error back to client:  5050 " + ex);
            }
         }
         else { con.init(); }
      }
      try
      {
         System.out.println("Closing server socket.");
         serverSocket.close();
      }
      catch (IOException e)
      {
         System.err.println("Could not close server socket. " + e.getMessage());
      }
   }
}
```

```
// The connection handler class - by Derek Molloy

import java.net.*;
import java.io.*;
import java.util.*;


public class HandleConnection
{

   private Socket clientSocket;                          // Client socket object
   private ObjectInputStream is;                         // Input stream
   private ObjectOutputStream os;                        // Output stream
   private SortService theSortService;

     // The constructor for the connecton handler
   public HandleConnection(Socket clientSocket)
   {
      this.clientSocket = clientSocket;
      theSortService = new SortService();
   }

   /** Thread execution method */
   public void init()
   {
      String inputLine;

      try
      {
         this.is = new ObjectInputStream(clientSocket.getInputStream());
         this.os = new ObjectOutputStream(clientSocket.getOutputStream());
         while (this.readCommand()) {}

       }
       catch (IOException e)
       {
            e.printStackTrace();
       }
   }

   /** Receive and process incoming command from client socket */
   private boolean readCommand()
   {
      Vector v = null;

      try
      {
         v = (Vector)is.readObject();
      }
      catch (Exception e)
      {
         v = null;
      }
      if (v == null)
      {
         this.closeSocket();
         return false;
      }

     System.out.println("Received: "+v.toString());
```

```
      theSortService.setData(v);
      this.getSorted();

        return true;
    }

    private void getSorted()
    {
        Vector returnData = theSortService.sortData();
        this.send(returnData);
    }

    // Send a message back through the client socket
    private void send(Object o)
    {
        try
        {
            System.out.println("Sending " + o);
            this.os.writeObject(o);
            this.os.flush();
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }

    // Send a pre-formatted error message to the client
    public void sendError(String msg)
    {
        this.send("error:" + msg);          //remember a string IS-A object!
    }

    // Close the client socket
    public void closeSocket()               //close the socket connection
    {
        try
        {
            this.os.close();
            this.is.close();
            this.clientSocket.close();
        }
        catch (Exception ex)
        {
            System.err.println(ex.toString());
        }
    }
}



// The DateTimeService that provides the current date
// by Derek Molloy.

import java.util.*;

public class SortService
{
      Vector v;
```

```
	public SortService()
	{
		v = new Vector();
	}

	public Vector sortData()
	{
		Object o[] = v.toArray();

		Arrays.sort(o);

		Vector x = new Vector();
		for (int i=0; i<o.length; i++)
		{
			x.add(o[i]);
		}

		return x;
	}

	public void setData(Vector v)
	{
		this.v = v;
	}
}
```

[16 marks]