SEMESTER ONE EXAMINATIONS 2007
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) Passing an object by constant reference allows us to pass a large block of data that cannot be modified without the need to copy the contents into a temporary variable, like in pass by value. The method/function cannot modify the contents of the memory passed by constant reference.

1(a ii) Destructors are used to tidy up objects on their destruction. For example in the case of a Bank Account object the destructor could print a paper copy of the account object before it is destroyed. Typically destructors are used in C++ to deallocate memory that was allocated by an object. Importantly destructors must be virtual as when the object is dynamically bound to a pointer the destructor most closely associated with the dynamic type must be call otherwise there will be incorrect deallocation when bound to a pointer.

1(a iii) The Object class in Java is actually the parent class of all classes. It defines methods that are common to all classes. It allows very complex operations to be performed in the Java language such as transporting objects across a network, storage on a common stack or file storage using standard methods. It is referenced as java.lang.Object.

1(a iv) (*prtToObject).someMethod();

1(a vi) Definition means "make this variable or method" here, allocating storage for the name. Declaration is the introduction of a name (identifier) to the compiler. It notifies the compiler that a method or variable of that name has a certain form, and exists somewhere.

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) **[12 marks total – 2 for each part i to vii]**

- 1(b) 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.

```
 7  class Account {
 8      public:
 9              float balance; // for demonstration only!
10              virtual ~Account(){}
```

```
};
class Current: public Account {};
class Test { public: float x, y;};

// Demo Function
void someFunction(int& c)
{
  c++;   // c can be modified
  cout << "c has value " << c << endl; //will output 11
};

int main()
{
   float f = 200.6f;
    // narrowing conversion, but we have notified the compiler
   int x = static_cast<int>(f);
   cout << x << endl;

   Account *a = new Current; //upcast - cast derived to base
   //type-safe downcast - cast base to derived
   Current *c = dynamic_cast<Current*>(a);

   const int i = 10;     // note i is const
   //someFunction(i);    // is an error as someFunction
                         // could modify the value of i
   someFunction(*const_cast<int*>(&i));
      // Allow i be passed to the function but it will still remain at 10.
   cout << "i has value " << i << endl; // will still output 10

   a = new Account;
   a->balance = 1000;
    //convert account address to long
   long addr = reinterpret_cast<long>(a);

   // safe to convert long address into an Account
   Account* b = reinterpret_cast<Account*>(addr);
   cout << "The balance of b is " << b->balance << endl;

   // could convert to any class regardless of
   // inheritance - ok this time! (not safe)
   Current* cur = reinterpret_cast<Current*>(addr);
   cout << "The balance of cur is " << cur->balance << endl;

   // works, but not definitely not safe this time!
   Test* test = reinterpret_cast<Test*>(addr);
   cout << "The value of the float x is " << test->x <<
       " and float y is " << test->y << endl;
}
```

Whole segment not required, only a shorter example like this.

[8 marks – 2 for each cast – 1 desc, 1 example]

1(c) In Java we have public, private and protected access specifiers, but we also have another access specifier "package". This is the default access specifier and means that all states and methods are accessible to all classes within the same package. There is no package access specifier keyword, it is simply the default if public, private or protected are not used. Access specifiers in Java are:

public - accessible everywhere, an interface method (same as C++)

private - accessible only in the class where it was defined (same as C++)

protected - accessible in the class where it is defined, the derived classes and in the same package (almost the same as C++ with the exception of the package)

"package" - default (there is no package specifier keyword) and means that it is accessible by any class in the same package. You can equate this to the C++ friendly condition by saying that all of the classes in the same package (directory) are friendly.

[5 marks]

2(a) (No code provided in advance)

```
package derek;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Calculator extends JFrame implements ActionListener{

    private JTextField screen;
    private String[] buttonStrings = {
            "7", "8", "9", "/",
            "4", "5", "6", "*",
            "1", "2", "3", "-",
            "0", "C", "+", "="
    };
    private float a;
    private int operator;

    Calculator()
    {
        super("EE553 Calculator");
        screen = new JTextField("");
        screen.setHorizontalAlignment(JTextField.RIGHT);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(BorderLayout.NORTH, screen);
        JPanel buttonpanel = new JPanel(new GridLayout(4,4));
        this.getContentPane().add(BorderLayout.CENTER, buttonpanel);
        for(int i=0; i<16; i++)
        {
            JButton temp = new JButton(buttonStrings[i]);
            temp.setActionCommand(buttonStrings[i]);
            temp.addActionListener(this);
            buttonpanel.add(temp);
        }
        this.pack();
        this.setVisible(true);
    }

    private float getScreenValue()
    {
        try{
            String temp = this.screen.getText();
            return Float.valueOf(temp).floatValue();
        }
        catch(Exception e) { return 0.0f; }
```

```
    }
    private void setScreenValue(float f) { this.screen.setText(Float.toString(f)); }
    private void clearScreen() { this.screen.setText(""); }
    private void appendScreen(String s)
    {
        String temp = this.screen.getText();
        this.screen.setText(temp.concat(s));
    }
    private void operatorPressed(int operator)
    {
        a = this.getScreenValue();
        this.operator = operator;
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getActionCommand().equals("C"))
        {
            this.a = 0.0f;
            this.operator = 0;
            this.clearScreen();
        }
        else if(e.getActionCommand().equals("/")) {this.operatorPressed(1);}
        else if(e.getActionCommand().equals("*")) {this.operatorPressed(2);}
        else if(e.getActionCommand().equals("+")) {this.operatorPressed(3);}
        else if(e.getActionCommand().equals("-")) {this.operatorPressed(4);}
        else if(e.getActionCommand().equals("="))
        {
            float result = 0;
            float b = this.getScreenValue();
            switch(this.operator)
            {
                case 1: result = a/b; break;
                case 2: result = a*b; break;
                case 3: result = a+b; break;
                case 4: result = a-b; break;
                default: result = a;
            }
            a = result;
            operator = 0;
            this.setScreenValue(result);
        }
        else {
            if (operator!=0) this.clearScreen();
            appendScreen(e.getActionCommand()); //must be numbers 0-9
        }
    }

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

}
```

[20 marks]

Approx 6 marks for coding, 5 marks for mouse handlers etc. 5 marks for functionality, 4 marks misc and understanding.

2(b)

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) ***(Code in italics was provided)***

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

class Account
{
  protected:

     float balance;
     int accountNumber;
     string owner;
     static int nextAccountNumber;

  public:
     Account(float, string);
     Account(float);
     Account();
     virtual void display();
     virtual bool makeWithdrawal(float &);
     virtual void makeLodgement(float &);

  /* Solution */
  private:
     void construct(float, string);
};

int Account::nextAccountNumber = 12345;

class CurrentAccount: public Account
{
     float overdraftLimit;

   public:

     CurrentAccount(float, string, float);
```

```
    CurrentAccount(Account, float);
    virtual void display();
    virtual bool makeWithdrawal(float &);
};


/* Solution */

void Account::construct(float bal, string name)
{
    accountNumber = nextAccountNumber++;
    balance = bal;
    owner = name;
}
Account::Account(float bal, string own){ construct(bal, own);}
Account::Account(float bal) { construct(bal, "Not defined");}
Account::Account()          { construct(0.0f, "Not defined");}

void Account::display()
{
    cout << "Account Number " << accountNumber << " with balance "
        << balance << " and owner " << owner << endl;
}

void Account::makeLodgement(float &amt)
{
    balance+=amt;
}

bool Account::makeWithdrawal(float &amt)
{
    if (amt<=balance)
    {
      balance-=amt;
      return true;
    }
    else return false;
}

CurrentAccount::CurrentAccount(float bal, string own, float limit):
  Account(bal, own), overdraftLimit(limit) {}
CurrentAccount::CurrentAccount(Account a, float limit):
  Account(a), overdraftLimit(limit) {}

void CurrentAccount::display()
{
    Account::display();
    cout << "And overdraft limit: " << overdraftLimit << endl;
}

bool CurrentAccount::makeWithdrawal(float &amt)
{
    if (amt<=(balance+overdraftLimit))
    {
      balance-=amt;
      return true;
    }
    else return false;
}
```

```
int main()
{
   Account a(100.0f, "Derek Molloy");
   CurrentAccount b(200.0f, "John Doe", 500.0f);
   CurrentAccount c(a, 1000.0f);

   a.display();
   b.display();
   c.display();

   system("pause");
}
```

[15 marks – ~1-2 marks for each method]

NB: Addition of construct() method, calling parent display() method and proper withdrawal method, and proper child constructors important in this answer.

Q3(b)

```
int main()
{

   vector<Account*> vStore;
   vStore.push_back(new Account(100.0f, "Derek Molloy"));
   vStore.push_back(new CurrentAccount(200.0f, "John Doe", 500.0f));
   vStore.push_back(new CurrentAccount(Account(250.0f, "Jack Black"), 1000.0f));
   for (int i=0; i<vStore.size(); i++)
      vStore[i]->display();
   system("pause");
}
```

[5 marks]

Q3(c) In C++ normal access control has the keywords:

- private - only accessible within the class.
- protected - only accessible within the class and its derived classes.
- public - accessible anywhere.

A class can declare a method to be a friend, allowing that method to access private and protected members (friendship is granted! Not acquired.). For example:

```
class SomeClass
{
  private:
    int x;
    friend void someMethod(SomeClass &);

  public:
    // the interface
};

// This method is somewhere!
void someMethod(SomeClass &a)
{
  a.x = 5;    //allowed!
}
```

It is important to note that someMethod() is not a member method of the class SomeClass. i.e. It does not have scope within the class SomeClass.
So if we tried:

```
SomeClass b;
b.someMethod();
```

This is illegal, as someMethod() is not a member method of the SomeClass class (and it is not public).
Friend Methods are useful as:

- Friend methods avoid adding unnecessary public methods to the interface.
- Prevent states from having to be made public.

However, the overuse of friend methods can make the code very complex and hard to follow.
We can add all the methods of a class as friends of another:

```
class A
{
  int x;
    friend class B;
  public:
    //methods here
};
```

Friendship is not inherited:

```
class B
{
  x(A &a)
  {
    a.x++;  //fine
  }
};

class C: public B
{
  y(A &a)
  {
    a.x++;  //illegal
  }
};
```

Friendship is not transitive:

```
class AA
{
  friend class BB;
};

class BB
{
  friend class CC;
};

class CC
{
  // Not a friend of AA
};
```

Question 4

4(a) The Math class is defined as final and has no public constructor. There is no requirement to create an instance of this class as it does not have any states. All the methods in the class are static so that they can be called without an object – e.g.
Math.sqrt(25)
int x = (int) (Math.random()*99.9999) + 1;

It is important to add 1 otherwise 100 will never be reached.

[5 marks, 3 for code]

4(b)

```
// The Calendar Solution

 import javax.swing.*;
 import java.awt.*;
 import java.awt.event.*;
 import java.util.*;

 public class DateApplication extends JFrame implements ActionListener
 {

        private JPanel panel = new JPanel();
        private JButton calculate;
        private JTextField day, year, dotw;
        private JComboBox month;
        private Object[] months = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
        private String[] dotwstr = {"Null", "Sunday", "Monday", "Tuesday",
                        "Wednesday", "Thursday", "Friday", "Saturday" };

        public DateApplication()
        {
          super("EE553 Date Application");

          JPanel controls = new JPanel(new FlowLayout());

          controls.add(new JLabel("Day:"));
          this.day = new JTextField(4);
          controls.add(this.day);

          controls.add(new JLabel("Month:"));
          this.month = new JComboBox(months);
          controls.add(this.month);

          controls.add(new JLabel("Year:"));
          this.year = new JTextField(6);
          controls.add(this.year);

          this.calculate = new JButton("Calculate");
          this.calculate.addActionListener(this);
          controls.add(this.calculate);

          controls.add(new JLabel("Day of the week:"));
          this.dotw = new JTextField(10);
          this.dotw.setEnabled(false);
          controls.add(this.dotw);

          this.getContentPane().add(controls);

          this.pack();  //set the size of the frame according
                        //to the component's preferred sizes
          this.setVisible(true);    //make the frame visible
        }

        public void actionPerformed(ActionEvent e)
        {
                if(e.getSource().equals(calculate))
```

```
                {
                  try{
                    int yeari = (new Integer(this.year.getText())).intValue();
                    int monthi = this.month.getSelectedIndex();
                    int dayi = (new Integer(this.day.getText())).intValue();
                    GregorianCalendar c = new GregorianCalendar(yeari, monthi, dayi);
                     this.dotw.setText(dotwstr[c.get(Calendar.DAY_OF_WEEK)]);
                }
                catch(Exception ex)
                  {
                     this.dotw.setText("Error");
                  }
                }
        }

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

[20 marks]

[6 for user-interface, 8 for Calendar work, 6 for events and type conversions]

Question 5.

5(a)

In C++ methods are non-virtual by default, so to replace the behaviour (allow over-riding) of a method in the derived class you have to explicitly use the virtual keyword in the parent class. You are able to replace the behaviour of a non-virtual method but this causes serious difficulties when the behaviour of the object depends on the static rather than the dynamic type! In Java, methods are virtual by default and we always operate on the dynamic type of the object. You cannot specify a method as non-virtual, but there is a final keyword. Once you use the final keyword on a method you cannot replace it - so there are no issues with static and dynamic types.

In Java we have the facility to define a class inside of another class. We refer to this class as an *inner class*. The only issue worth mentioning with inner classes is that they have full access to the private data and methods of the class in which they are nested. Because of this, we do not need to pass state pointers to methods that need callback.

In Java all methods must be defined within a class. There are no global methods or variables. This might seem unreasonable, but it works well. For example, the mathematical routine square root can be accessed through the Math class - a static class that allows you to call methods like Math.sqrt(25.0); allowing us to even replace mathematical operations, perhaps with different precision operations, by importing our own mathematical class.

The Java garbage collector manages all memory in Java. It reclaims all memory for objects once there are no remaining references to that object. When the garbage collector runs, it searches for all objects that have no references and reclaims the memory.

C++ developers are required to provide their own memory managment routines, otherwise a complex C++ application, running for a long period would simply cause the computer to run out of memory. Implementing memory managment correctly in C++ is complicated, and so memory leaks are a very frequent problem with C++ developed applications. Garbage collection was not added to the C++ language specification, as all developers would be required to deal with a standard garbage collector and every C++ application would incur the overhead. There are many 3rd party C++ garbage collectors available, such as the Boehm-

Demers-Weiser conservative garbage collector, that replaces the C malloc() or C++ new calls (It can alternatively be used as a leak detector!).

[8 marks – 2 marks each]

5(b)

```
import java.io.*;

public class DepositAccount implements Serializable
{
        private String owner;
        private float balance;
        private int number;

        public DepositAccount(String owner, float balance, int number)
        {
          this.balance = balance;
          this.owner = owner;
          this.number = number;
        }

        public void display()
        {
          System.out.println("Account with number:" + number);
          System.out.println(" with owner: " + owner);
          System.out.println(" and with balance: " + balance + "\n");
        }

        public float addInterest(float rate)
        {
          float interest = this.balance * rate / 100;
          this.balance = this.balance + interest;
          return interest;
        }
}
```

Client.java

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

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

public class Client
{

   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 Client(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 getInterest()
{
                DepositAccount d = new DepositAccount("Derek Molloy", 1000000, 12345);
                System.out.println("Present Deposit Account:");
                d.display();
                System.out.println("Sending account...");

                this.send(d);

                d = (DepositAccount)receive();
                if (d != null)
    {
                        System.out.println("Updated Deposit Account:");
                        d.display();
    }
                else { System.out.println("Error occurred!"); }
}

// 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[])
    {
            if(args.length>0)
            {
                Client theApp = new Client(args[0]);
                try
            {
               theApp.getInterest();
                }
              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);
    }
}
```

ConnectionHandler.java

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

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


public class ConnectionHandler
{

    private Socket clientSocket;                    // Client socket object
    private ObjectInputStream is;                   // Input stream
    private ObjectOutputStream os;                          // Output stream
    private DepositAccount d;

            // The constructor for the connecton handler
    public ConnectionHandler(Socket clientSocket)
    {
        this.clientSocket = clientSocket;
    }

    /** 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()
{
    d = null;

    try
    {
        d = (DepositAccount)is.readObject();
    }
    catch (Exception e)
    {
        d = null;
    }
    if (d == null)
    {
        this.closeSocket();
        return false;
    }
    else{

        this.applyInterest();
    }

    return true;
}

private void applyInterest()
{
       System.out.println("Applying Interest at 5%");
    float interest = d.addInterest(5.0f);
    System.out.println("Giving " + interest + " euro");
    this.send(d);
}

// 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 Server does not need to change.

[17 marks]
[7 marks for Deposit Account]
[5 marks for the client]
[5 marks for the connection handler]