

Question 1.

(a) Answer the following short questions. Keep your answers concise.

- (i) How is *scope resolution* performed in C++?
- (ii) Why are *destructors* required in C++?
- (iii) Explain how Java avoids the difficulties associated with *multiple inheritance*?
- (iv) Explain the use of the *Object class* in Java.
- (v) In C++ what is a *static local variable* and why would it be used?
- (vi) What does the term *overloading* mean?
- (vii) In the following piece of code explain what occurs. What is the value of x after execution (i.e. after the last line)? Why?

```
int x=6, *p;
p = &x;
*p+=++x;
```

[14 marks]

[2 marks each part]

1(a)(i)

- An **automatic variable** is a variable with a scope local to the block of code in which it was declared. Generally automatic variables are declared within functions, destroyed when the function ends.
- An **external variable** is used to declare a variable that is defined in some other module. (e.g. use: **extern float x;**)
- A **static variable** is used to make a variable private to the module in which it occurs, when used with the definition of a global variable.

The scope level can be resolved by the rules of scope, otherwise if required a variable at a different level can be used by using the scope resolution operator :: - for example Car.draw() is a method that could be called from a derived class.

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. This does not refer to states, but rather memory that was dynamically allocated using pointers and the new keyword. The destructor has the same name as the class and looks like Account::~~Account(). It does not accept any parameters, or return any values.

1(a)(iii)

Java uses Interfaces to avoid the use of multiple inheritance. 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 – no states.

- Methods in an interface are public and abstract.
- Instance variables in an interface are public, static and final.

1(a)(iv)

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)(v)

A static local variable is a variable that has memory allocated to it when it is first reached during programme execution. It only has local scope but exists until the program has executed to completion. An example of a static local variable (also known as a class variable) would be the nextAccountNumber state of the Account class, as this state is made available to all objects of the Account class.

1(a)(vi)

Overloading – Re-using the same method name with different arguments and possibly a different return type is known as overloading. In unrelated class there is no issues when re-using names but in the same class, we can overload methods, provided that those methods have different argument types. A method may not be distinguished by return type alone.

1(a)(vii) x will have a value of 14

the pointer points at the address of x and on the last line states that the value of $p = p + (++x)$ which increments x before the assignment to 7 but p points directly at x and so also has a value of 7 therefore $p = 7 + 7 = 14$

(b) What is an **abstract class**? Why would you create an abstract class?

[4 marks]

An Abstract class is a class that is incomplete.

- It cannot be instantiated
- It can only be used through inheritance
- It defines a behaviour that must be present in the child classes
- It is defined in C++ using the =0 assignment
 - E.g. `void char* getName() =0`
- In Java it is defined using the abstract keyword

You could use this to impose a behaviour on the child classes that you can still use within the base class. A good example of its use is the mouseEvents in Java. You could use abstract classes to impose a behaviour on a child, or another example would be the thread class.

(c) Discuss **constructors** in C++. Can they be overloaded? Why can they not be virtual? In what order are they called when inheritance takes place? What is the copy constructor and how can it provide specific functionality?

[7 marks]

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.

Question 2.

- (a) Explain the use of **friend functions** in C++. Why are they a useful feature? What difficulties could arise with the use of friend functions?

[8 marks]

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 function to be a friend, allowing that function to access private and protected members (friendship is granted! Not acquired.)

Example:

```
class aClass
{
    int x;
    friend void someFunction(aClass &a);
public:
    // the interface
};

// This function is somewhere!
void someFunction(aClass &a)
{
    a.x = 5;        //allowed!
}
```

It is important to note that **someFunction()** is not a member function of the class aClass. (It does not have scope within the class aClass)

So if we tried:

```
aClass b;
b.someFunction();
```

This is illegal, as someFunction() is not a member function of aClass.

- Friend functions avoid adding unnecessary public methods to the interface.
- Prevent us having to make member variables public.
- The overuse of friend functions can make the system very complex.

Friendship is not transitive.

Friendship is not inherited.

- (b) You will find a section of code on the disk. It contains the outline definition for the **Person**, **Student**, **Staff**, **Lecturer** and **Postgraduate** classes.

- Add constructors for the classes
- Add display() methods to the classes
- Create an array of Person objects and add constructed Lecturer and Postgraduate objects to the array.
- Loop through the array and call the display() methods of the objects – where the correct display() method should be called automatically.

[17 marks]

(b) The code given on the disk is:

```
/* A student record sytem for DCU
   written by Derek Molloy
   29th October 2002 */

#include <iostream>

class Person
{
    private:
        char* name;
        char* idNumber;
    public:
};

class Student: public Person
{
    private:
        char* course;
    public:
};

class Staff: public Person
{
    private:
        float salary;
        int phoneNumber;
    public:
};

class Lecturer: public Staff
{
    private:
        char* researchArea;
        char* module;
    public:
};

class Postgraduate: public Student
{
    private:
        Lecturer supervisor;
    public:
};

void main(void)
{
    cout << "Question 2 Application";
}
```

This can be modified to:

```
/* A student record sytem for DCU Solution
   written by Derek Molloy
   29th October 2002 */
```

```
#include <iostream>

class Person
{
    private:
        char* name;
        char* idNumber;
    public:
        Person(char* theName, char* theIDNumber);
        virtual void display();
};

Person::Person(char* theName, char* theIDNumber)
{
    name = theName;
    idNumber = theIDNumber;
}

void Person::display()
{
    cout << "Has name: " << name << " and ID number: " << idNumber << endl;
}

class Student: public Person
{
    private:
        char* course;
    public:
        Student(char* theName, char* theIDNumber, char* theCourse);
        virtual void display();
};

Student::Student(char* theName, char* theIDNumber, char* theCourse):
    Person(theName,theIDNumber), course(theCourse) {}

void Student::display()
{
    cout << " Is a student \n";
    Person::display();
    cout << " With a course: " << course << endl;
}

class Staff: public Person
{
    private:
        float salary;
        int phoneNumber;
    public:
        Staff(char*,char*,float,int);
        virtual void display();
};

Staff::Staff(char* theName, char* theIDNumber, float theSalary, int thePhoneNumber):
    Person(theName,theIDNumber), salary(theSalary),
    phoneNumber(thePhoneNumber) {}

void Staff::display()
{

```

```

        cout << "Is a staff member. \n";
        Person::display();
        cout << "With Salary of: " << salary << " and phone number: " << phoneNumber <<
endl;
    }

class Lecturer: public Staff
{
    private:
        char* researchArea;
        char* module;
    public:
        Lecturer(char*,char*,float,int,char*,char*);
        virtual void display();
};

Lecturer::Lecturer(char* theName, char* theIDNumber, float theSalary, int thePhoneNumber,
char* theResearchArea, char* theModule):
    Staff(theName, theIDNumber, theSalary, thePhoneNumber),
    researchArea(theResearchArea),
    module(theModule){}

void Lecturer::display()
{
    cout << "\nObject Is a Lecturer \n";
    Staff::display();
    cout << "With Research area: " << researchArea << " and module: " << module <<
endl;
}

class Postgraduate: public Student
{
    private:
        Lecturer supervisor;
    public:
        Postgraduate(char* theName, char* theIDNumber, char* theCourse, Lecturer
theSupervisor);
        virtual void display();
};

Postgraduate::Postgraduate(char* theName, char* theIDNumber, char* theCourse, Lecturer
theSupervisor):
    Student(theName,theIDNumber, theCourse), supervisor(theSupervisor) {}

void Postgraduate::display()
{
    cout << "\n\nIs a postgraduate\n";
    Student::display();
    cout << "And research Supervisor: " << endl;
    supervisor.display();
}

void main(void)
{
    cout << "Question 2 Application \n\n";

    Lecturer derek("Derek Molloy", "94971056", 10000, 5355, "Vision", "EE553");
    //derek.display();
    Postgraduate jack("Jack Murphy", "90079825", "EPPD1", derek);

```

```

//jack.display();

//For the last section I created an array of only 2 elements - completes spec.
Person *p[2] = {&derek, &jack};
for (int i=0; i<2; i++)
    p[i]->display();
}

```

```

/cygdrive/c/My Documents/My Teaching/EE553 (2002-2003)
$ a
Question 2 Application

Object Is a Lecturer
Is a staff member.
Has name: Derek Molloy and ID number: 94971056
With Salary of: 10000 and phone number: 5355
With Research area: Vision and module: EE553

Is a postgraduate
Is a student
Has name: Jack Murphy and ID number: 90079825
With a course: EEPD1
And research Supervisor:

Object Is a Lecturer
Is a staff member.
Has name: Derek Molloy and ID number: 94971056
With Salary of: 10000 and phone number: 5355
With Research area: Vision and module: EE553

molloyd@PORTALM /cygdrive/c/My Documents/My Teaching/EE553 <2002-2003>
$ _

```

[6 marks for the constructors]
 [5 marks for the display methods]
 [3 marks for the creation of the array]
 [3 marks for looping through the array]
[17 marks total]

Question 3

(a) What is **Remote Method Invocation** (RMI) and how is it used in Java? Explain the terms *skeletons* and *stubs*. What are the limitations of RMI?

[9 marks]

Remote method invocation (RMI) is a full-grown architecture for distributed computing, scalable to very complex tasks. Serialization handles the details of writing object level protocols, so the programmer can send objects to a stream without worrying about their structure. RMI provides a way of distributing objects as services so that a remote service request looks like a local request. The object is stationed in one place and the VM is responsible for serving the object declared exportable and puts it where an RMI object server can call on it when a request comes in. Once the object is exported, RMI takes care of the rest:

- Serialization
- Object transport
- Exception handling
- Security management.

Advance knowledge of the remote calls is essential. This involved the use of a pair of classes called **skeletons** and **stubs**. These two classes are derived directly from the

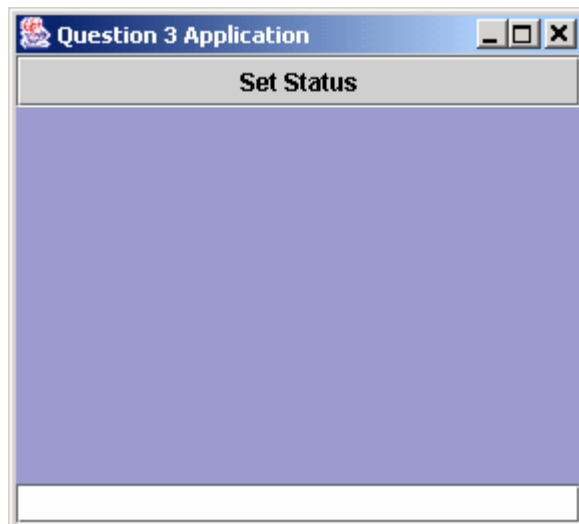
class file of the remote object. When the remote client calls on the object, it does so by a lookup. This lookup is coded into the client, as are the calls to the object's methods. If the lookup succeeds the RMI server returns the remote object's stub, which is a stand-in for the remote object's class and methods. On a call to any of one of these methods, the stub sends the request to the skeleton reference, which resides on the server side. The skeleton retrieves the operations of the method to co-ordinate routing the object's response back through the stub.

[9 marks- 2 each for stubs and skeletons – 5 for remaining description]

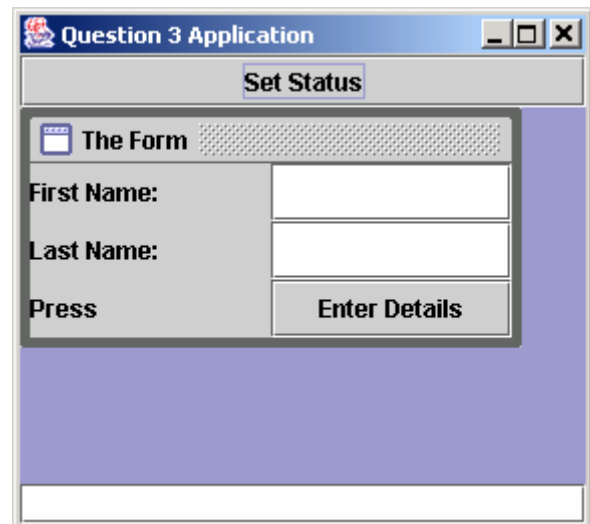
(b) Write a Java application that looks like the application below. The application:

- Should open up with the format as shown in (a)
- When the "Set Status" button is pressed the internal frame should appear as in (b)
- When the details are entered in the fields and "Enter Details" is pressed as in (c)
- The Status at the bottom should display the data entered in the fields as in (d)
- If the "Set Status" button is pressed again then it should begin again.

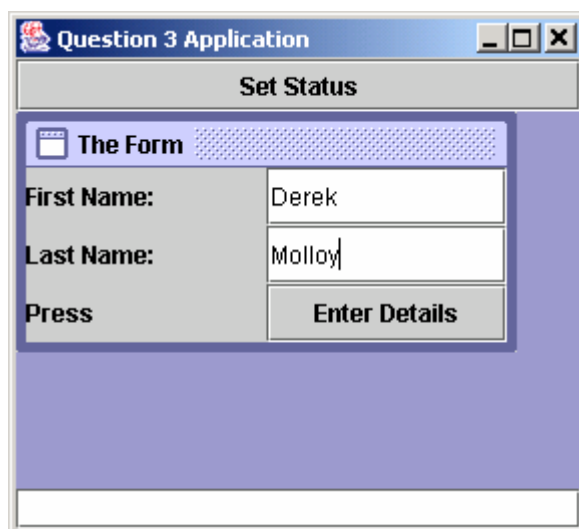
[17 marks]



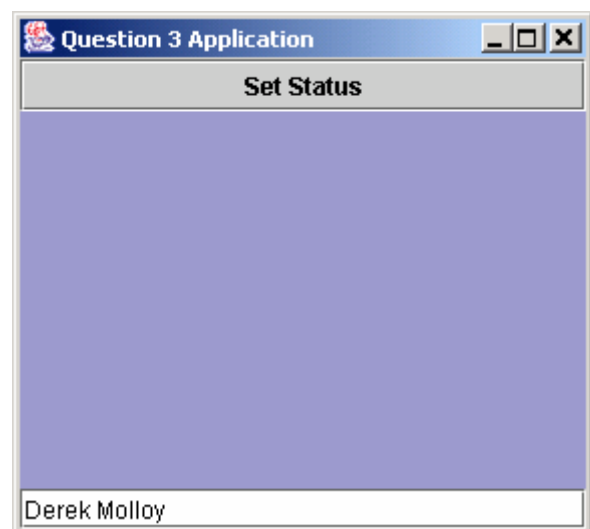
(a)



(b)



(c)



(d)

Code for The Application Itself:

```
package mypackage1;

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

public class Question3 extends JFrame implements ActionListener
{
    private JDesktopPane theDesktop;
    private MyForm theForm;
    private JButton newForm;
    private JTextField theStatus;

    public Question3()
    {
        super("Question 3 Application");
        this.getContentPane().setLayout(new BorderLayout());
        this.theDesktop = new JDesktopPane();
        this.newForm = new JButton("Set Status");
        this.newForm.addActionListener(this);
        this.theStatus = new JTextField();

        this.getContentPane().add("North", this.newForm);
        this.getContentPane().add("Center", this.theDesktop);
        this.getContentPane().add("South", this.theStatus);

        this.setSize(400,400);
        this.show();
    }

    public void setStatus(String s)
    {
        theStatus.setText(s);
    }

    public void actionPerformed(ActionEvent e)
    {
        this.theForm = new MyForm(this);
        this.theDesktop.add(this.theForm);
    }

    public static void main(String args[])
    {
        new Question3();
    }
}
```

Code for the Internal Dialog:

```
package mypackage1;

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

```

import java.awt.event.*;

public class MyForm extends JFrame implements ActionListener
{
    private String firstName;
    private String lastName;
    private JTextField firstField;
    private JTextField lastField;
    private JButton enterName;
    private Question3 theApp;

    public MyForm(Question3 q3)
    {
        super("The Form");
        theApp = q3;

        this.firstField = new JTextField();
        this.lastField = new JTextField();
        this.enterName = new JButton("Enter Details");
        this.enterName.addActionListener(this);

        this.getContentPane().setLayout(new GridLayout(3,2));
        this.getContentPane().add(new JLabel("First Name:"));
        this.getContentPane().add(firstField);
        this.getContentPane().add(new JLabel("Last Name:"));
        this.getContentPane().add(lastField);
        this.getContentPane().add(new JLabel("Press"));
        this.getContentPane().add(enterName);

        this.setSize(250,120);
        this.show();
    }

    public void actionPerformed(ActionEvent e)
    {
        this.firstName = firstField.getText();
        this.lastName = lastField.getText();
        theApp.setStatus(this.firstName + " " + this.lastName);
        this.setVisible(false);
        this.dispose();
    }
}

```

Important points for marks:

- Develop it as two classes
- Very important to pass the return value to the application
- Lay out the forms
- Implements the action events

An Outline mark allocation (Can change depending on student's implementation)

[5 marks for knowledge to create internal frames]

[5 marks for passing value to main window]

[3 marks for layout of main window]

[3 marks for action event implementation]

[16 marks total]

Question 4.

(a) The `java.util.Math` contains many mathematical operations. In your opinion, why is there no public constructor for the `Math` class? Show an example of how you would use the `random()` method to pick a random whole number between 1 and 100 (inclusive).

[5 marks]

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.

(b) Write a Java application that uses a Java Vector and Stack as follows:

- Create an `Account` class that stores the account number, account balance, account owner and has a constructor, `display()`, `makeLodgement()` and `makeWithdrawal()` methods.
- In a command line Java application create a `Vector` object
- Store 3 anonymous `Account` objects in this vector
- Create a loop, looping through the `Vector` object and display the details of the `Account` objects stored in this vector.
- In the same loop “push” these elements onto a `Stack` object.
- From the `java.util.Stack` API documentation work out how to extract the elements from the stack and display the `Account` details.

[15 marks]

The `Account` class should look like this:

```
package mypackage1;

public class Account
{
    private String owner;
    private float balance;
    private int number;

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

    public void makeLodgement(float amount)
    {
        this.balance+= amount;
    }

    public boolean makeWithdrawal(float amount)
    {
        if (amount > this.balance) return false;
    }
}
```

```

        balance-=amount;
        return true;
    }

    public void display()
    {
        System.out.println("Account details are - Owner: " + owner + " has balance " + balance + "
and act. number " + number);
    }
}

```

The Application Class should look like this

```

package mypackage1;

import java.util.*;

public class VectorTest
{
    public VectorTest()
    {
    }

    public static void main(String args[])
    {
        Vector v = new Vector();
        Stack s = new Stack();

        v.add(new Account("Derek", 100, 1234));
        v.add(new Account("Tom", 50, 1235));
        v.add(new Account("Jack", 500, 1236));

        for (int i=0; i<v.size(); i++)
        {
            Account temp = (Account) v.elementAt(i);
            temp.display();
            s.push(temp);
        }
        while(!s.empty())
        {
            Account temp = (Account) s.pop();
            temp.display();
        }
    }
}

```

The important points in this application are:

- Creation of the Account class – approx 3 marks
- Ability to create anonymous objects – approx 2 marks
- Knowledge of the Vector class – approx 2 marks
- This questions assumes that the student does not know the java.util.Stack class, and looks up the documentation in the exam, working out how to use the class – 5 marks
- Tests that the student is able to cast convert. – approx 2 marks
- Displays knowledge of objects of the Object class. – approx 1 marks

[15 marks total]

- (c) In the example in (b) you created an Account object. How would you compare two Account object to see if they had the same values? Show this with a short segment of code.

[5 marks]

In this question it is important to note that:

```
Account a = new Account("Derek",100,1234);
Account b = new Account("Derek",100,1234);
if(a.equals(b)) System.out.println("Same");
```

Does not work as the object references are not the same! We would have to write a method in the Account class to do this comparison. If we override the equals() method then we can add a valid comparison. Add the code as below for the equals() method and add getXXX methods – or make the states public (bolded code below)

Account Class

```
package mypackage1;

public class Account
{
    private String owner;
    private float balance;
    private int number;

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

    public void makeLodgement(float amount)
    {
        this.balance+= amount;
    }

    public boolean makeWithdrawal(float amount)
    {
        if (amount > this.balance) return false;
        balance-=amount;
        return true;
    }

    public float getBalance() { return this.balance; }
    public int getNumber() { return this.number; }
    public String getOwner() { return this.owner; }

    public boolean equals(Account a)
    {
        if (this.balance!=a.getBalance()) return false;
        if (this.number!=a.getNumber()) return false;
        if (!this.owner.equals(a.getOwner())) return false;
        return true;
    }

    public void display()
    {
```

```

    System.out.println("Account details are - Owner: " + owner + " has balance " + balance + "
and act. number " + number);
}
}

```

Question 5.

- (a) Explain using an example when you would need to synchronize a segment of code when using Java threads? (Your answer should show a line-by-line step through of a segment of code, explaining why it would not work correctly if the segment of code was not synchronized). If synchronization is a solution to making an application thread safe, then as programmers, why should we not just synchronize all code?

[7 marks]

It can be difficult to control threads when they need to share data. It is difficult to predict when data will be passed, often being different each time the application is run. We add synchronization to the methods that we use to share this data like:

public synchronized void theSynchronizedMethod()

or we can select a block of code to synchronize:

synchronized(anObject)

```

{
}

```

This works like a lock on objects. When two threads execute code on the same object, only one of them acquires the lock and proceeds. The second thread waits until the lock is released on the object. This allows the first thread to operate on the object, without any interruption by the second thread.

First Thread	Second Thread
call theSynchronizedMethod()	
acquires the lock on the theObject	
executes theSynchronizedMethod on theObject	
	calls theSynchronizedMethod on theObject
	some other thread has a lock on theObject
returns from theSynchronizedMethod()	
	acquires the lock on the theObject
	executes theSynchronizedMethod on theObject

Again, synchronization is based on objects:

- two threads call synchronized methods on different objects, they proceed concurrently.
- two threads call different synchronized methods on the same object, they are

synchronized.

- two threads call synchronized and non-synchronized methods on the same object, they proceed concurrently.

Static methods are synchronized per class. The standard classes are multithread safe.

It may seem that an obvious solution would be to synchronize everything!! However this is not that good an idea as when we write an application, we wish to make it:

- Safe - we get the correct results
- Lively - It performs efficiently, using threads to achieve this liveliness

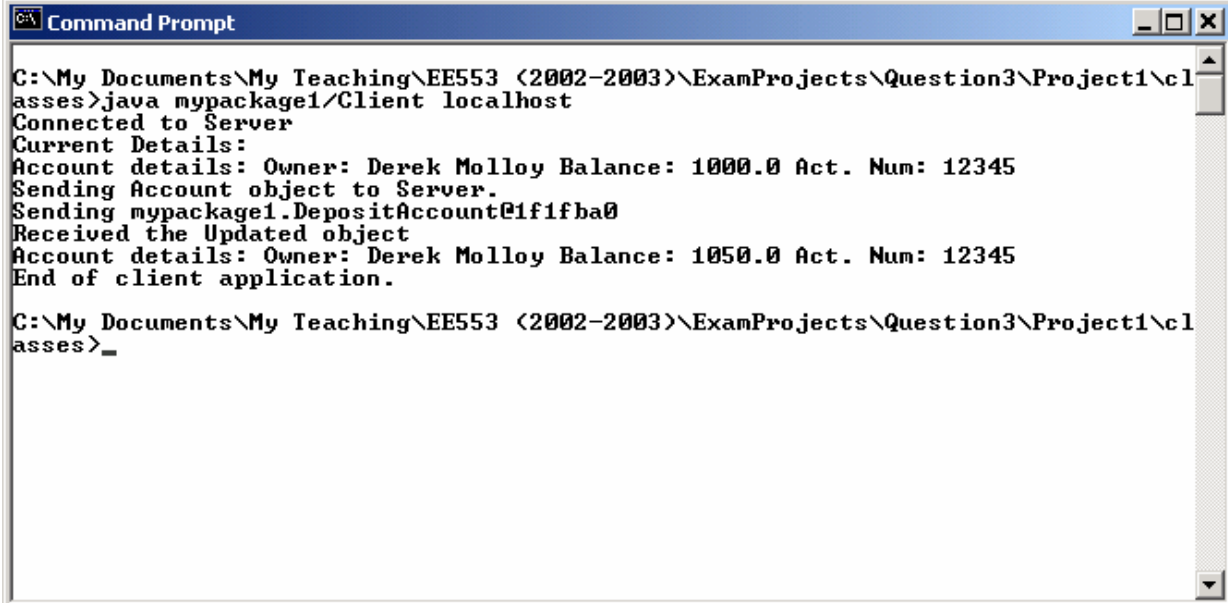
These are conflicting goals, as too much synchronization causes the program to execute sequentially, but synchronization is required for safety when sharing objects. Always remove synchronization if you know it is safe, but if you're not sure then synchronize.

- (b) Write a Java client/server application pair, where the client passes a DepositAccount object to the server and the server calculates the interest on the account using the current interest rate available on the server. The account object is then passed back to the client, where the client displays the details including the updated balance.

You have been supplied with three sets of code to handle the basic aspects of this application. These are called:

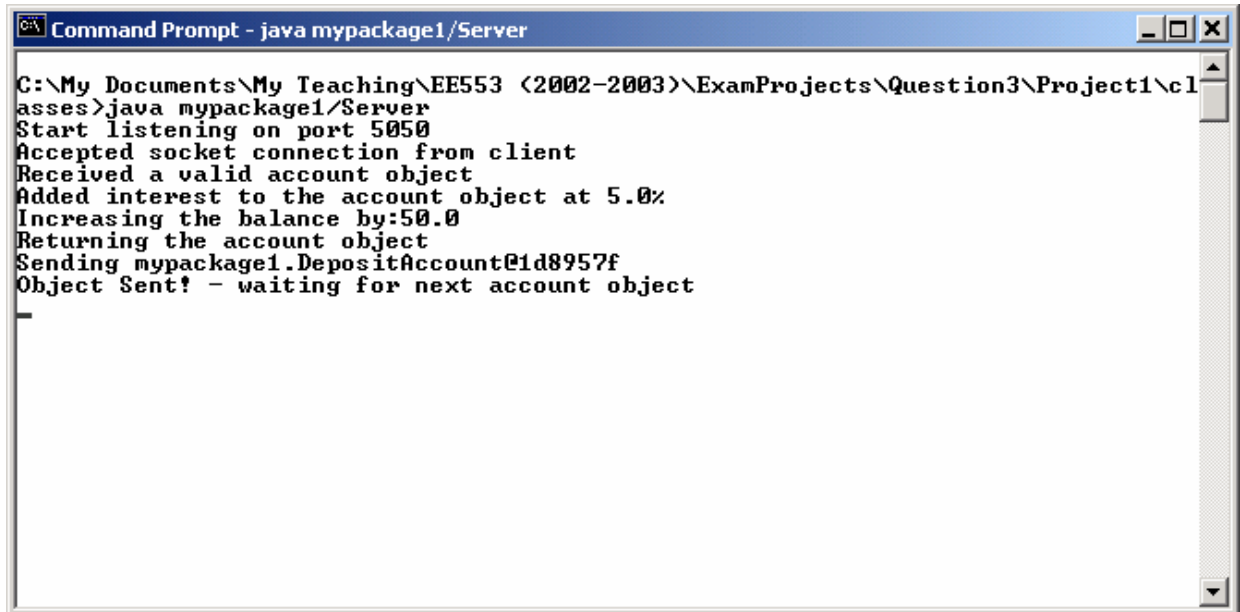
- **Client.java**,
- **Server.java** and
- **ConnectionHandler.java**

These files are in the directory **question5**.



```
Command Prompt
C:\My Documents\My Teaching\EE553 (2002-2003)\ExamProjects\Question3\Project1\classes>java mypackage1\Client localhost
Connected to Server
Current Details:
Account details: Owner: Derek Molloy Balance: 1000.0 Act. Num: 12345
Sending Account object to Server.
Sending mypackage1.DepositAccount@1f1fba0
Received the Updated object
Account details: Owner: Derek Molloy Balance: 1050.0 Act. Num: 12345
End of client application.

C:\My Documents\My Teaching\EE553 (2002-2003)\ExamProjects\Question3\Project1\classes>
```

```

C:\My Documents\My Teaching\EE553 (2002-2003)\ExamProjects\Question3\Project1\classes>java mypackage1/Server
Start listening on port 5050
Accepted socket connection from client
Received a valid account object
Added interest to the account object at 5.0%
Increasing the balance by:50.0
Returning the account object
Sending mypackage1.DepositAccount@1d8957f
Object Sent! - waiting for next account object

```

Classes Required

Deposit Account class – must implement serializable and must add in a calculate interest method - **[approx 8 marks]**

Client Class – must be modified to send account objects **[approx 5 marks]**

Connection Handler class – must be modified to call methods of deposit account **[approx 5 marks]**

Server Class – does not have to change from server class given.

[18 marks total]

Solution Code:

DepositAccount Class

```

package mypackage1;

import java.io.Serializable;

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

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

    public void makeLodgement(float amount)
    {
        this.balance+= amount;
    }

    public boolean makeWithdrawal(float amount)

```

```

    {
        if (amount > this.balance) return false;
        balance-=amount;
        return true;
    }

    public void display()
    {
        System.out.println("Account details: Owner: " + owner + " Balance: " + balance + " Act.
Num: " + number);
    }

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

```

ClientClass

```

package mypackage1;
// The Date 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", 1000, 12345);
        System.out.println("Current Details:");
        d.display();
        System.out.println("Sending Account object to Server.");
        this.send(d);
        d = (DepositAccount)receive();
        if (d != null)
        {
            System.out.println("Received the Updated object");
            d.display();
        }
        System.out.println("End of client application.");
    }

    // 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 Class

```

package mypackage1;
// 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

    // The constructor for the connecton handler
    public ConnectionHandler(Socket clientSocket)
    {
        this.clientSocket = clientSocket;
        //Set up a service object to get the current date and time
        //theDateService = new DateTimeService();
    }

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

        try
        {
            this.is = new ObjectInputStream(clientSocket.getInputStream());
            this.os = new ObjectOutputStream(clientSocket.getOutputStream());
            while (this.readAccount()) {}
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }

    /** Receive and process incoming command from client socket */
    private boolean readAccount()
    {
        DepositAccount tempAct = null;
        float rate = 5;
    }
}

```

```

    try
    {
        tempAct = (DepositAccount)is.readObject();
        System.out.println("Received a valid account object");
    }
    catch (Exception e)
    {
        tempAct = null;
    }
    if (tempAct == null)
    {
        this.closeSocket();
        return false;
    }

    // invoke the appropriate function based on the command
    if (tempAct!=null)
    {
        float amount = tempAct.addInterest(rate);
        System.out.println("Added interest to the account object at "+rate+"%");
        System.out.println("Increasing the balance by:"+amount);
        System.out.println("Returning the account object");
        this.send(tempAct);
        System.out.println("Object Sent! - waiting for next account object");
    }
    else
    {
        this.sendError("Invalid Account Object");
    }
    return true;
}

// 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());
    }
}
}
```

Exam Details (General) 2002-2003 Dublin City University, Ireland.

MODULE: Object Oriented Programming – EE553

COURSE: M.Eng./Grad. Dip. in Electronic Systems
M.Eng./Grad. Dip. in Telecommunications Engineering
RAEC – Remote Access to Continuing Eng. Education

YEAR: Postgraduate (Year 5)

EXAMINER: Dr. Derek Molloy (DCU extension 5355)

TIME ALLOWED: 3 hours

INSTRUCTIONS: Answer FOUR questions. All questions carry equal marks