Question 1.

(a) **[14 marks - 2 per section]** Explaining the terms:

(i) The keyword protected in Java means that the method or state has visibility within the package where the class is defined and in any derived class. In C++ the keyword protected means that the method or state has visibility within that class and its derived classes.

(ii) An event listener allows an application to listen for events. For example an action listener can be used to listen for button presses or enter presses within a textField. Events are generated when the button is pressed or when the button is pressed. You can use for example myTextField.addActionListener(this) to add a listener to a component.

(iii) Almost every class within Java extends the Object class. This allows us to treat these classes in a common way. It also provides a common set of methods that may be used on all Classes, such as compareTo(Object o). This form is very useful when sending instances of Objects across a network.

(iv) Scope resolution is useful in C++ when we wish to reference a variable that is outside our scope. For example: within the currentAccount class that extends the Account Class we could write a method currentAccount::display() that calls the method Account::display() in the base class.

(v) The ResultSet Object is created after a JDBC query is executed on a database. It contains lines that contain the results of the database query. We can step through the result set and extract the columns of the query.

(vi) JINI is a version of Java that runs on the Java K-Virtual Machine, a minimal version of a JVM. It is aimed at consumer devices, such as fridges, cookers etc., where a very small amount of processing power is available.

(vii) The implements keyword allows us to use an interface within our class. An Interface defines a common set of functionality that our Class must implement. This allows us to define a common behaviour for classes. An example of this is the use of the mouse.

(b) **[6 marks – 2 per paragraph]** An abstract class is a class that defines methods that do not have implementation, i.e. it is incomplete. It cannot be instantiated, and can only be used through inheritance (i.e. polymorphism).

It is used to define a common behaviour in derived classes. We can write a method in the base class that actually uses a method that is defined in the derived class, but has not as yet been implemented.

An example use is the case of the generic class Car. We could define a method draw() that is abstract. In the derived class SportsCar we implement this method, and a similar method for the other derived class SaloonCar. These draw() method do different things depending on which derived class is used.

(c) **[5 marks – ~1 per point]** A virtual machine has several advantages when developing Internet based applications:

- ?? It allows us to develop platform independent bytecode applications.
- ?? It provides security management, preventing hostile applets from attacking the hard disk, memory or locking up the CPU.
- ?? It allows compact applications to be developed
- ?? Any platform that can develop this virtual machine can run the applications.

Question 2.

(a) **[9 marks – 3 per paragraph]** Java provides an Image class (java.awt.Image) for the purpose of storing images (extends Object). This class contains useful methods for obtaining information about the image object. It has been a part of the Java SDK since the original release. Some of the methods associated with the Image class include getWidth(), getHeight(), getSource(), and getGraphics(). An image object can be obtained from a source, such as the Applet.getImage() method and then displayed on the canvas using the drawImage() method which is a member function of the Graphics class. This method allows the scaling of an image using a bounding box that can be used for performing basic magnification, using the Image class method.

An image consists of information in the form of pixel data, in the case of a Java image object; a pixel is represented by a 32-bit integer. This integer consists of four bytes containing an alpha (transparency) value, a red intensity value, a green intensity value and a blue intensity value, so each colour and the transparency level are integer values in the range 0 to 255. This representation of pixel data is referred to as the default RGB colour model.

We can draw lines, circles and rectangles directly onto this image by obtaining the graphics context of an image (theImage.getGraphics())and then using that to draw lines, circles etc.. For example, g.drawLine(0,0,100,100) draws a line between the point (0,0) and (100,100).

(b) **[16 marks – 1 (example1) 7 (queston2b.java) 8(TheFilter.java)]** The question involves writing three files.

Example1.html

```
<title> Exam paper question 2 </title>

<body>
<hr>
<applet code=Question2B.class width=400 height=300></applet>
<hr>
</body>
```

Question2b.java

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

// An applet to load an Image and Filter it

public class Question2B extends Applet {
   Image theImage;

   public void init()
   {
      // load the image 'test.jpg' in the current directory (URL)
      theImage = this.getImage(this.getDocumentBase(),"question2.gif");

       // Wait for the image to load
       while(theImage.getHeight(this)==-1){/*do nothing*/};

        TheFilter filter = new TheFilter();
        this.theImage = Toolkit.getDefaultToolkit().createImage(
              new FilteredImageSource(this.theImage.getSource(),filter));
   }

   public void paint(Graphics g)
   {
      g.drawImage(this.theImage,10,10,this);
   }
}
```

TheFilter.java

```
import java.awt.image.RGBImageFilter;

class TheFilter extends RGBImageFilter {

   public TheFilter()
   {
      // Not dependent on x,y postion => faster
      canFilterIndexColorModel = true;
   }

   public int filterRGB(int x, int y, int rgb)
   {
      int r = (rgb >> 16) & 0xff;
      int g = (rgb >> 8) & 0xff;
      int b = (rgb >> 0) & 0xff;

              r = 255 - r;
              g = 255 - g;
              b = 255 - b;

      return (rgb & 0xff000000) | (r<<16) | (g<<8) | (b<<0);
   }
}
```

Gives the output window:

Question 3.

(a) **[5 marks – 3 text, 2 code]** 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
•A constructor must not have a declared return value (not even void!)
•A constructor cannot be virtual.

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.
•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!

(b) **[11 marks – 4 per type, 3 for code] Multiple inheritance** is one aspect of C++ that leads to complications, especially when a defined class inherits from two classes.

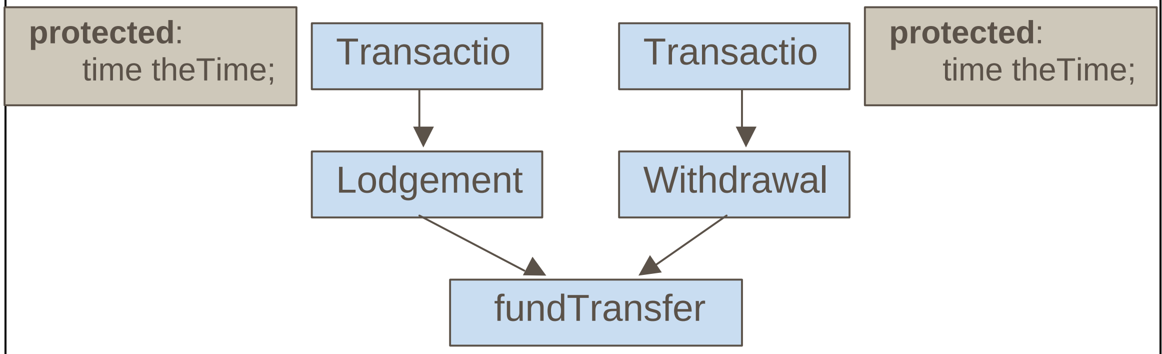


Fig Q3(b)-1 – An example where multiple base class objects are required.

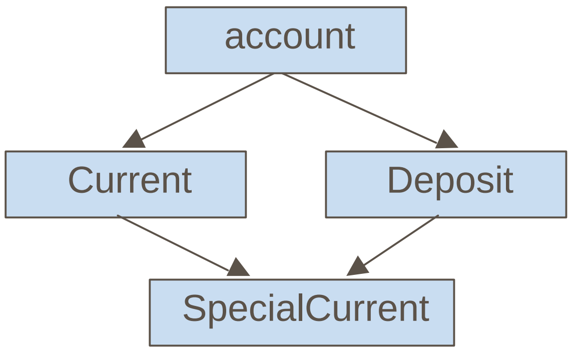


Fig Q3(b)-2 – An example where multiple base class objects are not required.

In Q3(b)-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(b)-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.

(c) **[9 marks – 3 per code eg.]** C++ supports separate compilation, i.e. changes to one class, do not require the re-compilation of the other classes. **This means that it is good practice to place each class in a separate source file!** This also has the effect of increasing the clarity of the code.In actual fact, each class should be stored in two files:
• a source file (.cpp) – the implementation of the functions
• a header file (.h) – the definition of the class
So in **account.h** put:

```
// account.h

listingclass account{
    protected:
        int theAccountNumber;
        char* theOwner;
        float theAccountBalance;
    public:
        account(int theNumber, char* theOwner);
        virtual void makeLodgement(lodgement &amt);
        virtual void makeWidthdrawal(withdrawal &amt);
};
```

and in **account.cpp** put:

```
// account.cpp listing

#include "account.h" // use "" for user defined include
account::account(int theNumber, char* theOwner){
    //add code
}
account::makeLodgement(lodgement &amt){
    //add code.
}   //etc…
```

```
// currentAccount.h listing

#ifndef currentAccount_h  //check not already defined
#define currentAccount_h //if not, define!
#include "account.h"     //include the account header
class currentAccount: public account{
    protected:
        float theOverdraftLimit;
    public:
        currentAccount(int theNumber, char* theOwner,
          float theOverdraftLimit);
        virtual void makeWithdrawal(withdrawal &amt);
        // and any others…
};
#endif  // currentAcount_h
```

Question 3.

(a) [8 marks – 2 per type, 4 for keywords] Extending the Thread class The first way is to extend the Thread class to create our own Thread class.

```
// Extend the Thread class
public class Counter extends Thread
{
        public void run()
        {
                ....
        }
}
```

This method overrides the Thread.run() method for its own implementation. The run() method is where all the work of the Counter class thread is performed.
Implementing the **Runnable** interface
The other way is implementing the Runnable interface.

```
// Extend the Thread class
public class Counter implements Runnable
{
        Thread t;
        public void run()
        {
                ....
        }
}
```

We have an instance of the Thread class as a variable of the Counter class.
Implementing Runnable provides greater flexibility in the creation of the class counter as we still have the facility to extend some other class.
Suspending and resuming a thread

Once we use the stop() method on a thread, we cannot use the start() method, as we have terminated the operation of the thread. We can pause the execution of the thread for a certain amount of time using the sleep() method, however we cannot use it to pause for an undeterminable amount of time. For this we can use the suspend() and resume() methods.

They work like this:
Java has thread scheduling that monitors all running threads in all programs and decides which thread should be running. There are two main characteristic of thread:

?? Priority - are normal, user-defined threads.

?? **Daemon** - are low priority threads (often called service threads) that provide basic service to programs, when the load on the CPU is low. The garbage collector thread is an example of a daemon thread. It is possible for use to convert a user thread into a daemon thread, or vice-versa, using the setDaemon() method.

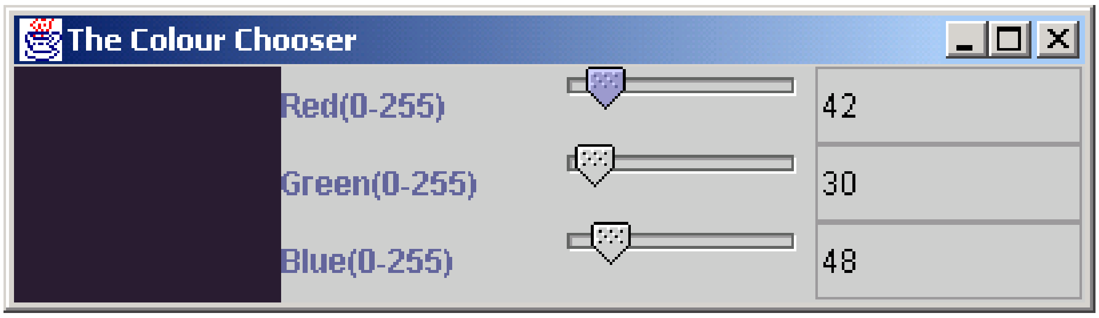
Thread **Priorities**: The scheduler decides which thread should be running based on a priority number assigned to the thread. The priority number has a value between 1 and 10. There are three pre-defined priorities:

Thread.MIN_PRIORITY - has the value of 1
Thread.NORM_PRIORITY - has the value of 5
Thread.MAX_PRIORITY - has the value of 10

Synchronization 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){ }

(c) **[17 marks – even for interface and implementation]** Here is the code JcolorChooser.java:

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



public class JColorChooser extends JFrame implements ChangeListener
{
	private JSlider reds,greens,blues;
	private JTextField redt,greent,bluet;
	private JPanel theOutput;

	public JColorChooser(String theTitle)
	{
		super(theTitle);
	}

	private void readSliders()
	{
		int red = this.reds.getValue();
		int green = this.greens.getValue();
		int blue = this.blues.getValue();

		this.redt.setText(""+red);
		this.greent.setText(""+green);
		this.bluet.setText(""+blue);

		Color c = new Color(red,green,blue);
		this.theOutput.setBackground(c);
	}

	public void init()
	{
		this.getContentPane().setLayout(new GridLayout(1,4));
		theOutput = new JPanel();
		theOutput.setBackground(Color.black);
		this.getContentPane().add(theOutput);

		reds = new JSlider(JSlider.HORIZONTAL,0,255,0);
		greens = new JSlider(JSlider.HORIZONTAL,0,255,0);
		blues = new JSlider(JSlider.HORIZONTAL,0,255,0);
		reds.addChangeListener(this);
		blues.addChangeListener(this);
		greens.addChangeListener(this);
		redt = new JTextField("0");
		greent = new JTextField("0");
		bluet = new JTextField("0");
		redt.setEditable(false);
		greent.setEditable(false);
		bluet.setEditable(false);
		JPanel LabelGrid = new JPanel(new GridLayout(3,1));
		LabelGrid.add(new JLabel("Red(0-255)"));
		LabelGrid.add(new JLabel("Green(0-255)"));
		LabelGrid.add(new JLabel("Blue(0-255)"));
		this.getContentPane().add(LabelGrid);

		JPanel SliderGrid = new JPanel(new GridLayout(3,1));
		SliderGrid.add(reds);
		SliderGrid.add(greens);
		SliderGrid.add(blues);
		this.getContentPane().add(SliderGrid);

		JPanel FieldGrid = new JPanel(new GridLayout(3,1));
		FieldGrid.add(redt);
		FieldGrid.add(greent);
		FieldGrid.add(bluet);
		this.getContentPane().add(FieldGrid);

		this.pack();
		this.show();
	}

	public void stateChanged(ChangeEvent e)
	{
		this.readSliders();
	}


	public static void main(String[] args)
	{
		JColorChooser jc = new JColorChooser("The Colour Chooser");
		jc.init();
		jc.show();
	}

}
```



Question 5.

(a) **[3 marks – total]** Before the advent of object serialization, if the programmer wished to send objects over a network, it was necessary to break up the objects into its constituent parts, ints, arrays etc. Object serialization makes this a relatively simple task. Once an object is committed to a stream its state is fixed. The usual means of communicating through streams is a procedural one, the client sends a request, and the server sends a response. Serialization alone requires the user to maintain the protocol for communicating.

(b) **[7 marks – total, 2 each for stubs and skeletons]** 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.

(c) **[15 marks – 5 per client, handler, service]** A Java client/server pair, where the client sends a string to server and returns an encrypted version. Here are the modifications to the code required. First off, write the encryption service:

```
public class EncryptService
{

	public String encryptString(String s)
	{
		String back = "";
		char[] theChars = s.toCharArray();
		for (int i=0; i<s.length(); i++)
		{
			theChars[i] = (char)((int)theChars[i] + 1);
		}
		back = new String(theChars);
		return back;
	}

}
```

Then modify the client.java to send the string as a parameter:

```
public void getEncrypt(String toEncrypt)
{
	String theEncrypt = "";
	System.out.println("Encrypt the string: "+toEncrypt);
	send("Encrypt"+toEncrypt);
	theEncrypt = (String)receive();
	if (theEncrypt != null)
	{
		System.out.println("The Encrypted string is " + theEncrypt);
	 }
}
```

and modifying the main method()

```
if(args.length>0)
{
	EncryptClient myApp = new EncryptClient(args[0]);
	try
	{
		myApp.getEncrypt(args[1]);
	}
	catch (Exception ex)
	{
		System.out.println(ex.toString());
	}
}
```

Then modify the ConnectionHandler.java code:

```
/** Receive and process incoming command from client socket */
private boolean readCommand()
 {
	String s = null;

	try
	{
	   s = (String)is.readObject();
	}
	catch (Exception e)
	{
	   s = null;
	}
```

```
    if (s == null)
    {
       closeSocket();
       return false;
    }

    // invoke the appropriate function based on the command
          String startx = s.substring(0,7);
    if (startx.equals("Encrypt"))
    {
       getEncrypt(s.substring(7,s.length()));
    }
    else
    {
       sendError("Invalid command -> " + s);
    }

    return true;
 }

 private void getEncrypt(String theString)
 {
    String encrypt = theService.encryptString(theString);
    send(encrypt);
 }
```

The Server.java class remains mainly the same.