



## DUBLIN CITY UNIVERSITY

REPEAT EXAMINATIONS 2009/2010

### [SOLUTIONS]

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

**COURSE:** MTC – M.Eng. in Telecommunications Engineering  
MEN – M.Eng. in Electronic Systems  
IPME - Individual Postgrad. Modules-Electronics  
MEQ - Masters Engineering Qualifier Course  
GCES – Grad Cert. in Electronic Systems  
GCTC – Grad Cert. in Telecommunications Eng.  
GDE – Graduate Dip. in Electronic Systems  
GTC – Graduate Dip. in Telecommunications Eng.

**YEAR:** C

**EXAMINERS:** Dr. Derek Molloy, Ext no. 5355

**TIME ALLOWED:** 3 Hours

**INSTRUCTIONS:** Answer Four Questions

#### Question 1(a)

(i) `this` - allows us to refer to "this object", i.e. to access states or methods for the class that the code is currently within. It allows us to pass a reference to the current object and it also allows us to access the states of the class that are currently out of scope.

(ii) The `Math` class has no constructor, which makes sense as there is no requirement to have an object of `Math`. The methods are called statically, e.g. `Math.sqrt(25.0)` which compares directly to a global function in C++. In Java it is not possible to have a global function.

(iii) Pointers require a dereference type – e.g. `int* x`; means that `x` has a dereference type of `int`. This is required as a pointer needs to know the size in order to increment its value – e.g. `x++`, which needs to know how many bytes to move in memory. Also operations like `cout << *x`; needs to know how to treat the value that has been dereferenced – in this case the output stream operator is being passed an `int` value.

(iv) Translator programs called compilers then convert the high-level languages into machine code. C, C++ and Java are all examples of high-level languages. Large programs can take significant time to compile from the high-level language form into the low-level machine code form. An alternative to this is to use interpreters; programs that execute high-level code directly by translating instructions on demand. These programs do not require compilation time, but the interpreted programs execute much more slowly.

(v) 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.

(vi) break causes a looping statement to quit and continue causes the statement to move to the next iteration.

(vii) A modal dialog box must be closed or exited before an application can continue, but a modeless dialog box can 'run in parallel' with the application. Modal dialog boxes are much easier to control in feeding information to the application, but a modeless dialog can return values at any time, which is more difficult to structure.

[1(a) 2 marks each part]

[14 marks total]

**Question 1(b)** In this segment of code class A is a friend of class B. This means that when an object of the class A is passed to the method x() within class B that class B has full access to the private x state of A and Point 1 is therefore valid. However, friendship is not inherited and class C is therefore not a friend of class A, meaning that it has no access to the private state x of class A.

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.

[6 marks]

**Question 1(c)**

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.

[5 marks]

**Question 2(a)**

STL can be described and categorised as follows:

Containers: Data structures with memory management capability.

Iterators: Logic that binds containers and algorithms together.

Algorithms: Provide a mechanism for manipulating data through the iterator interface.

Some examples of sequential containers are:

- vector: A flexible array of elements.
- deque: A dynamic array that can grow at both ends.
- list: A doubly linked list.
- map: A collection of name-value pairs, sorted by the keys value.
- set: A collection of elements sorted by their own value.
- multimap: Same as a map, only duplicates are permitted.
- multiset: Same as a set, only duplicates are permitted.

Some Iterators:

- Input iterators
- Output iterators
- Forward iterators
- Bidirectional iterators
- Random Iterators

The algorithms can be classified as:

- Non-modifying algorithms: e.g. for\_each, count, search ...
- Modifying algorithms: for\_each, copy, transform ...
- Removing algorithms: remove, remove\_if, ...
- Mutating algorithms: reverse, rotate, ...
- Sorting algorithms: sort, partial\_sort, ...
- Sorted range algorithms: binary\_search, merge ...
- Numeric algorithms: accumulate, partial\_sum ...

[ 7 marks – not all text required, only 4 examples of each]

[3 marks for general description, 4 marks for examples]

### Question 2(b)

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;
```

```
template <class T>
void outputFunction(T x)
{
    cout << x << endl;
}
```

```
int main(void)
{
    int x;
    vector<int> vect; // declare a vector container of ints

    cout << "Please enter a list of numbers:" << endl;
    while(cin >> x)
    {
```

```

        vect.push_back(x);
    }

    sort(vect.begin(), vect.end()); // sort the array

    cout << endl << "Sorted output of numbers:" << endl;

    // loop through vector with for_each loop and execute
    // outputFunction() for each element

    for_each(vect.begin(), vect.end(), outputFunction);
}

```

[6 marks, 2 for the template function, 1 for vector, 1 for sort, 2 for for\_each]

**Question 2(c)** 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 ambiguous 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.

[1 mark for static and const, 2 each for dynamic and reinterpret]

[6 marks total]

**Question 2(d)**

Example in the notes like this:

```

3 #include <iostream>
4 using namespace std;
5
6 // Demo classes with a virtual base class
7 class Account {
8     public:
9         float balance; // for demonstration only!
10        virtual ~Account(){}
11 };
12 class Current: public Account {};
13 class Test { public: float x, y;};
14
15 // Demo Function
16 void someFunction(int& c)
17 {
18     c++; // c can be modified

```

```

19 cout << "c has value " << c << endl; //will output 11
20 }
21
22 int main()
23 {
24     float f = 200.6f;
25     // narrowing conversion, but we have notified the compiler
26     int x = static_cast<int>(f);
27     cout << x << endl;
28
29     Account *a = new Current; //upcast - cast derived to base
30     //type-safe downcast - cast base to derived
31     Current *c = dynamic_cast<Current*>(a);
32
33     const int i = 10; // note i is const
34     //someFunction(i); // is an error as someFunction
35         // could modify the value of i
36     someFunction(*const_cast<int*>(&i));
37     // Allow i be passed to the function but it will still remain at 10.
38     cout << "i has value " << i << endl; // will still output 10
39
40     a = new Account;
41     a->balance = 1000;
42     //convert account address to long
43     long addr = reinterpret_cast<long>(a);
44
45     // safe to convert long address into an Account
46     Account* b = reinterpret_cast<Account*>(addr);
47     cout << "The balance of b is " << b->balance << endl;
48
49     // could convert to any class regardless of
50     // inheritance - ok this time! (not safe)
51     Current* cur = reinterpret_cast<Current*>(addr);
52     cout << "The balance of cur is " << cur->balance << endl;
53
54     // works, but not definitely not safe this time!
55     Test* test = reinterpret_cast<Test*>(addr);
56     cout << "The value of the float x is " << test->x <<
57         " and float y is " << test->y << endl;
58 }
59

```

1 mark for static, 1 mark for static and 2 marks each for reinterpret and constant  
[6 marks total]

### Question 3(a)

```

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

class Person{
    string name, id;
public:

```

```

        Person(string, string);
        virtual void display();
        virtual string getRole() = 0;
};

Person::Person(string nm, string anid): name(nm), id(anid) {}

void Person::display()
{
    cout << endl << "Role is: " << getRole() << endl;
    cout << "Name is: " << name << " and id is: " << id << endl;
}
class Student: public Person
{
    string programme;
    int year;
public:
    Student(string, string, string, int);
    virtual void display();
    virtual string getRole();
};

Student::Student(string name, string id, string prog, int yr):
    Person(name, id), programme(prog), year(yr) {}
string Student::getRole() { return "student"; }

void Student::display()
{
    Person::display();
    cout << "Programme is: " << programme << " and year is: " << year << endl;
}

class Lecturer: public Person
{
    string office;
    int phoneNum;
public:
    Lecturer(string, string, string, int);
    virtual void display();
    virtual string getRole();
};

Lecturer::Lecturer(string name, string id, string off, int ph):
    Person(name, id), office(off), phoneNum(ph) {}
string Lecturer::getRole() { return "lecturer"; }

void Lecturer::display()
{
    Person::display();
    cout << "Office is: " << " and phone number is: " << phoneNum << endl;
}

```

```

int main(void)
{
    // test of Student and Lecturer
    Student s("John", "1234", "MENG", 1);
    s.display();
    Lecturer l("Derek", "5123", "S356", 5355);
    l.display();
}

```

**[10 marks]**

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

### Question 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

### Question 3(c)

```

Storage<Person*, 10> store;
store.addElement(new Lecturer("Derek Molloy", "5123", "S356", 5355));
store.addElement(new Student("John Doe", "1234", "MENG", 1));
store.addElement(new Student("James Doe", "1235", "MENG", 1));
for (int i=0; i<=store.getSize(); i++)
    store[i]->display();

```

**[3 marks]**

### Question 3(d)

```

vector<Person*> vStore;
vStore.push_back(new Lecturer("Derek Molloy", "5123", "S356", 5355));

```

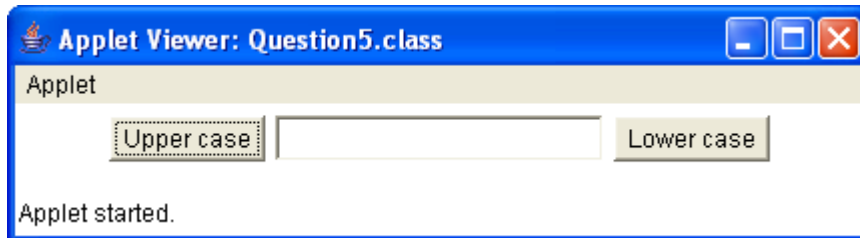
```

vStore.push_back(new Student("John Doe", "1234", "MENG", 1));
vStore.push_back(new Student("James Doe", "1235", "MENG", 1));
for (int i=0; i<vStore.size(); i++)
    vStore[i]->display();

```

[5 marks]

#### Question 4(a)



```

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>
```

[9 marks]

**Question 4(b)**

```
public class CaseApp extends Frame implements ActionListener {

    public CaseApp () {
        add constructor to replace init()
    }
    ...
    public void start() {

    }

    public void stop() {
        ...
    }

    public void run() {
        ...
    }

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

[4 marks]

[1 change add main, 1 can const, 1 call start, 1 frame and setsize, setvisible]

**Question 4(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 implements the interface it can be passed as an object to the Frame.

[4 marks for Interfaces and when used, 3 for this example]

[7 marks total]

#### Quesiton 4(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 total]

**Question 5(a)** 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.

[9 marks overall]

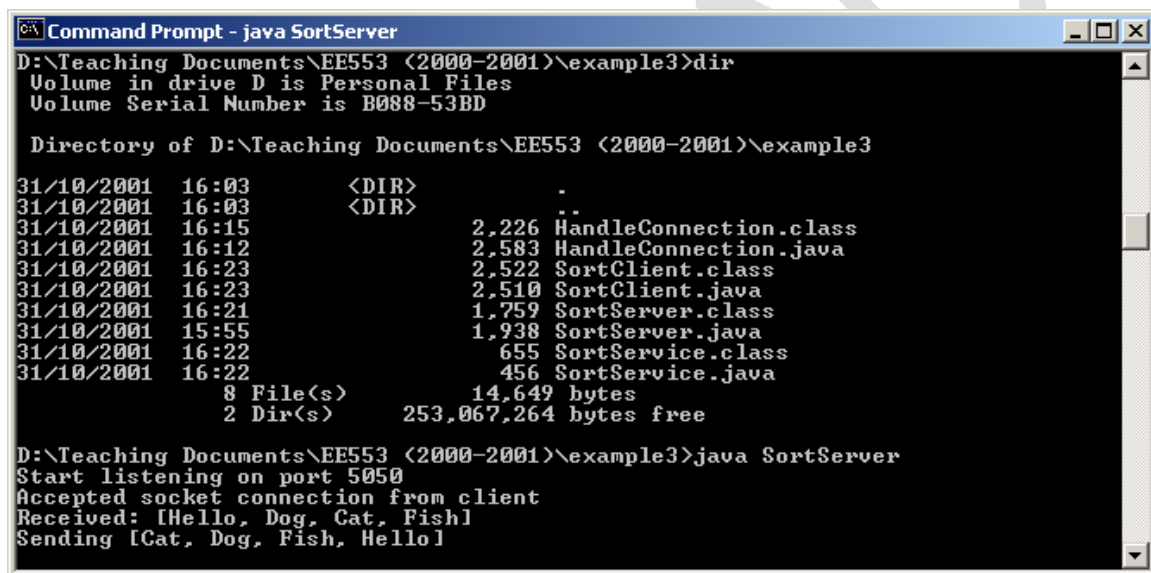
[4 marks for correct example]

[3 marks for intro explain]

[2 marks for safety vs. lively]

### Question 5(b)

Solution:

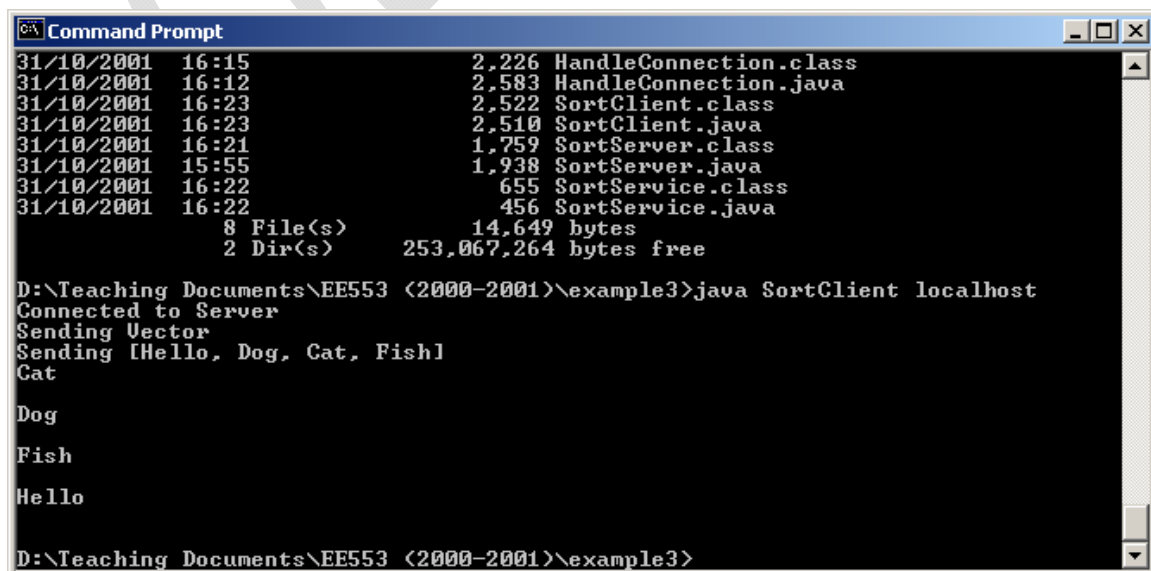


```
C:\> Command Prompt - java SortServer
D:\Teaching Documents\EE553 (2000-2001)\example3>dir
Volume in drive D is Personal Files
Volume Serial Number is B088-53BD

Directory of D:\Teaching Documents\EE553 (2000-2001)\example3

31/10/2001  16:03      <DIR>          .
31/10/2001  16:03      <DIR>          ..
31/10/2001  16:15             2,226  HandleConnection.class
31/10/2001  16:12             2,583  HandleConnection.java
31/10/2001  16:23             2,522  SortClient.class
31/10/2001  16:23             2,510  SortClient.java
31/10/2001  16:21             1,759  SortServer.class
31/10/2001  15:55             1,938  SortServer.java
31/10/2001  16:22             655  SortService.class
31/10/2001  16:22             456  SortService.java
               8 File(s)              14,649 bytes
               2 Dir(s)              253,067,264 bytes free

D:\Teaching Documents\EE553 (2000-2001)\example3>java SortServer
Start listening on port 5050
Accepted socket connection from client
Received: [Hello, Dog, Cat, Fish]
Sending [Cat, Dog, Fish, Hello]
```



```
C:\> Command Prompt
31/10/2001  16:15             2,226  HandleConnection.class
31/10/2001  16:12             2,583  HandleConnection.java
31/10/2001  16:23             2,522  SortClient.class
31/10/2001  16:23             2,510  SortClient.java
31/10/2001  16:21             1,759  SortServer.class
31/10/2001  15:55             1,938  SortServer.java
31/10/2001  16:22             655  SortService.class
31/10/2001  16:22             456  SortService.java
               8 File(s)              14,649 bytes
               2 Dir(s)              253,067,264 bytes free

D:\Teaching Documents\EE553 (2000-2001)\example3>java SortClient localhost
Connected to Server
Sending Vector
Sending [Hello, Dog, Cat, Fish]
Cat
Dog
Fish
Hello

D:\Teaching Documents\EE553 (2000-2001)\example3>
```

// 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 connection 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 total]

[6 marks for sort service]

[4 marks for constructing suitable client]

[4 marks for suitable server]

[2 marks for communications]