University of St Andrews
School of Computer Science

Appendix IV

Examples of Current Practical Assignments

CS2001 Foundations of Computation

Practical Week 6

CS2001 Practical Week 6
A Registry Database

This practical is worth **14%** of the overall coursework mark.

Aims and Objectives

The aim of the practical is to program data structures using the *java.util* classes.

The object of the practical is to program is to build a registry database of the inhabitants of the island of Java (no relation).

Learning Outcomes

By the end of this practical you should understand how to reuse a variety of library classes to as a foundation for creating data types.

Requirements

Specification

This practical continues the theme of the week 4 Practical. The JRO (Javanese Registry Office) wish to maitain and interrogate information about each member of the Javanese population. This will include their name, sex, date and time of birth, date and time of death if deceased, age, parents and children.

A person's name will act as their unique identifier, so the JRO wants the registry to be held as a map from name to other information. That is, they want it to implement the interface *java.util.Map*. Combining in the additional functionality required leads the JRO to specifying the following interface for the registry:

```
public interface JRO extends java.util.Map {
    // return a person's details
    // (or a suitable message if they don't exist)
    String finger( Person who );

    // return a person's age
    // (or a suitable message if they don't exist)
    String age( Person who );

    // register a baby boy with the given details
    void boy( String name, Male father, Female mother );

    // register a baby girl with the given details
    void girl( String name, Male father, Female mother );

    // register the death of Person p
    String death( Person who );

    // remove all the deceased who are at least expiry years old
    // return their names
    String bury( double expiry );

    String toString();
}
```

The JRO also specifies some other auxiliary interfaces mentioned in JRO.

A *Person* is an interface representing an individual member of the Javanese population, with subinterfaces *Male* and *Female* and interface *F1* for representing their offspring:

```
public interface Person {
    // get their date of name
    String name();

    // get their date of birth
    java.util.Date dob();

    // get their date of death
    java.util.Date dod();

    // Calculate their age
    int age();

    // Find all their children
    F1 children();

    // Are they alive?
    boolean alive();

    // Kill them
    void kill(java.util.Date dod );

    String toString();
}

public interface Male extends Person {
    String toString();
}

public interface Female extends Person {
    String toString();
}

public interface F1 extends java.util.List {
    String toString();
}
```

Implementation

The JRO not only wants the registry to implement the interface *java.util.Map*, but has also stipulated that it should be implemented as far as possible via the standard classes from the *java.util* library. Your task is to design such classes that implement the above interfaces.

Specifically, design the following classes:

- *Registry* that implements *JRO* and extends *HashMap*
- *Persons* that implements *Person*
- *Males* that implements *Male* and extends *Persons*
- *Females* that implements *Female* and extends *Persons*
- *Children* that implements *F1* and extends *LinkedList*

Remember that you can avoid the repetitions of *java.util* by including

```
import java.util*;
```

at the top of a class file.

*As a guideline, your program file should be around **200 lines long** excluding comments.*

Resources

- The class *Harness* at:

  resource/Modules/CS2001-FC/ practical s/Practical-W6/Harness.java

  provides the *main* method which implements an interactive user interface to the registry via these 1-letter commands:

*	invoke *Registry.toString* to list all the names in the registry.
?	read a name and call *finger*, using *Person.toString* to print the person's details;
@	read a name and call *finger*, using *Person.age* to print the person's age;
b	read appropriate details and call *boy*;
g	read appropriate details and call *girl*;
-	read a name and call *death*;
!	read an expiry age and call *bury*;
x	exit.

  Harness makes use of:

- The class *TIO* (in a package called *SimpleIO*) provides simple token I/O. You can also use its output methods *write/writeln* — they are just calls to *System.out.print/println*.

  Javadoc of *SimpleIO* can be found at:

  resource/Modules/CS2001-FC/java/SimpleIO/

  See also coments in the Week 5 Exercise handout.

- The standard classes:

  java.util.Date to keep track of dates of birth and death;

  java.util.Calendar to maintain a real time clock in the registry.

Deliverables

Hand in via MMS:

- Source code and documentation: report, user manual and test results.
- The report should describe what your program does and how it meets the requirements. [See the general instructions for further information on the format and content of reports.]

Marking

Component	Marks
Implementation of Registry and Person classes	6
Implementation of Man, Woman and Children classes	3
Documentation: report, user manual, test results	5
Total	**14 marks**

CS2002 Computer Architecture

Practical 1

University of St Andrews
School of Computer Science

CS2002 Computer Architecture Practical 1 **Semester 2, 2002–03**

The aim of this **unassessed** practical is to acquire experience with *[x-]spim* and a basic understanding of the MIPS assembly language; to do this, write (in several stages, starting from an existing program) a simple “encryption algorithm”, testing it with either the *spim* or the *x-spim* simulator.

For each part, hand in a documented solution. The percentages attached to the parts indicate the effort required.

Part 1 (20%):
The program "p1.mips" (attached, and available on *resource*) is an incorrect MIPS program to copy a block of words from one area of memory to another, terminating when a null word (i.e. all 0s) is read and copied.

In the language C, it might, as a function, be as follows:

```
copy_words(unsigned *source, unsigned *dest)
{
  while(*source != 0)
    {
      *dest++ = *source++;
    }
  *dest = 0;
}
```

Test this program and correct the error. If using *spim*, you will find it useful to work out the address of the label `dest` and see what data is at that address or nearby.

Hand in your evidence of testing the incorrect program, explanation of what is wrong, the corrected program and evidence for your belief that the corrected program is correct.

Part 2 (80%):
Modify the corrected program by adding code that reverses the order of bits in a word, so that the word `0x80000000` becomes `0x00000001`, the word `0xDEADBEEF` becomes `0xF77DB57B`, etc. In C it might, packaged as a function, be as follows:

```
unsigned reverse_word(unsigned w)
{
  int i;
  unsigned res = 0;
  for (i = 32; i != 0; --i)
    {
       res = (res << 1) ;      % shift res left a bit
       res = res | (w & 0x1); % 'or' res with last bit of w
       w   = (w >> 1);        % shift w right a bit
    }
  return(res);
}
```

The reversal should be done to words at the point between their being loaded and their being stored. There is no need to use a subroutine; in-line code will do

Hand in a well-commented print out of the modified program, including details of the use of additional registers, and commented evidence of testing.

RD (after KH)

The incorrect program "p1.mips" is as follows:

```
#       Register Table
# $v0     For system call code (in fact, it's register $2)
# $8      Source address; also called $t0
# $9      Destination address; also called $t1
# $10     Contents of source; also called $t2

     .text
     .globl main
main:     la   $8, source      # Set up source/dest
          la   $9, dest
loopo:    lw   $10, ($8)       # Get the next word
          beq  $10, $0, done   # Done if 0
          addi $8,4            # Increment the addresses
          addi $9,4            #    by 4 bytes each
          sw   $10, ($9)       # Store the word
          b loopo              # and repeat
done:     sw   $0, ($9)        # Store the terminating 0
          li $v0, 10           # Code for 'exit'
          syscall              # Exit

     .data
     .globl source
source:   .align 4             # Not nec. needed
     .word 0xABCDEF00          # Test Data
     .word 0x12345678          # Each word is 4 bytes
     .word 0xDEADBEEF
     .word 0xFEEDBACC
     .word 0

     .globl dest
dest:     .space 20            # 20 bytes for these 5
```

CS2003 Advanced Internet Programming

Practical 5

CS2003 Practical 5 (DOC 3): Viewing Dom trees using JavaScript 22nd March

Objectives

The objective of this practical is to gain a deep understanding of the Document Object Model and its manipulation via JavaScript

Assignment

Here is a sample HTML page whose source may be found here.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<META http-equiv=Content-Type content="text/html; charset=iso-8859-1">
</HEAD>
<BODY>
<SCRIPT src="colourNodes.js">
</SCRIPT>
<P>Here is a prog which displays the node names in the HTML DOM of this
document.</P>
<P>For those with technical knowledge, notice how HTML1.1 is not compatible with
XML and so the input close tag is treated as a separate DOM node.</P>
<P>HTML4.01 of course solves this problem....</P>
<FORM><INPUT onclick="showNodes( document.documentElement )" type=button value="Show
HTML node names">
</INPUT></FORM></BODY></HTML>
```

You are required to write a JavaScript DOM program that displays the DOM tree in a clear manner. My sample program (whose output is shown below) colours the nodes and displays the DOM node types.

```
element: {HTML}
    element: {HEAD}
        element: {TITLE}
        element: {META}content="text/html; charset=iso-8859-1"
    element: {BODY}
        element: {SCRIPT}
        element: {P}
            text: Here is a prog which displays the node names in the HTML DOM of th
        element: {P}
            text: For those with technical knowledge, notice how HTML1.1 is not comp
        element: {P}
            text: HTML4.01 of course solves this problem....{#text}
        element: {FORM}
            element: {INPUT}
                text: {#text}
```

Tips

Do the practical in stages:

1. Write a basic tree traverser using first child and next sibling and make sure that it works.
2. Embellish the output from the program a bit at a time – first indentation, then labels then colour etc.
3. If you do this the easy way you will write the output to a new document.
4. If you are feeling adventurous create a new DOM tree with the output.
5. If you get fed up with this and have found it ridiculously easy you can try doing the same practical using Xerces and Java.

Resources

1. Java Script reference manual on Resource: http://resource.dcs.st-and.ac.uk/JavaScript/jsref/index.htm
2. Dom Specification including JavaScript binding: http://resource.dcs.st-and.ac.uk/Modules/CS2003-AIP/JS-resources/DOM2-Core.pdf
3. Xerces manual on Resource: http://resource.dcs.st-and.ac.uk/Modules/CS2003-AIP/Xerces%20manual/

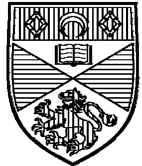
Hand In

You are required to hand in via MMS two files: a sample HTML file similar to the one above and a JavaScript file containing your DOM traverser.

RM 13th March 2002

CS3051 Software Engineering

Assignment 1 and 2



School of Computer Science

CS3051 : Systems Analysis Case Study

A G.P. System

Two medical practices share one building and are to merge their administration and booking systems, using one group of secretaries or administration assistants to assist both practices. Both their current computer systems are to be updated to allow the following functionality:

1. Booking an Appointment.
When a patient phones or calls at the surgery, the new system should allow the practice secretary to enter the patient's identification details, the practice they belong to and preferred appointment details. The system will respond with a range of possible dates and times of which one is booked.

2. Keeping the Appointment
When the patient arrives for their appointment the secretary marks the appointment as having been attended.

3. Doctor's Access to Patient Records.
Each doctor will have a computer in his/ her office at which they can view their own list of patients for the current surgery. A doctor can retrieve a patient's record from the computer system. This record will contain the patient's name, address, date of birth, a list of drug allergies and a list of all drugs that have been prescribed to that patient. Naturally a drug cannot be prescribed to a patient who is allergic to it.
When the patient leaves, the doctor updates the appropriate patient's record to indicate that they have been seen. This allows the secretary to know how many people the doctor has actually seen and how late they are running.

4. Current Drugs
The computer system will also contain a list of currently acceptable drugs to aid resource management by the practice. The drugs list is updated after the doctors have read of new ones in medical journals or have been visited by a pharmaceutical representative. Drugs are removed from the list when the government proscribes their usage or the drug is withdrawn by the pharmaceutical company or the doctors stop prescribing it over a period of time.

5. Assessment Tasks

1. Identify and list the functions that the system must provide for its users. (10%)

2. Identify the different user screens that have to be provided by the system. (10%)

3. Identify files, user identities and accessible objects or artefacts, such as databases, in the system. (10%)

4. Draw the Required Logical Data Flow Diagram (RLDFD), a level 2 DFD, for the G.P. system. (20%)

5. Create an Entity Life History (ELH) for the entity DRUGS - the list of currently acceptable drugs used by the practice. (10%)

6. Using Object Interaction Diagrams sketch out

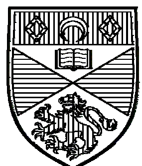
   (i) 2 'normal' or valid scenarios which you would expect the completed system to be able to achieve without difficulty. (20%)

   (ii) 1 exception scenario which the system should be able to catch.(10%)

   (iii) 1 invalid scenario which the system should be able to catch.(10%)

Hand drawn diagrams are acceptable as long as they are neat. All work should be accompanied by a top sheet with the student's name and ID number.

Due Date : end of week 5 Friday 31st October 4pm



University of St Andrews

School of Computer Science

CS3051 Software Engineering
Assignment 2 (2003)

You are required to write two short reports on academic papers from the field of Software Engineering; the first of no more than two pages, the second of no more than four pages. Both reports must be in the style of a technical report, i.e. the use of passive form, title, authors, abstract, keywords, sectioning and conclusion. The writing should be succinct, with a short discussion of each of the points made.

The reports should be submitted electronically with a cover sheet for each report including the name of the paper discussed, the authors of the paper and the statement "Report by <your name>" and the date of submission at the bottom of the page. The due time and date is 4 p.m. Friday 19th December 2003 (end of Week 12).

The form of the report itself is not fixed but should attempt to follow the template:

<Title of academic paper>
<Authors>
<Journal/ Conference>
<Date of paper>

<your keywords>
<your abstract/ overview>

<sections following your discussion or dissemination of the paper>
<conclusion – summarizing the authors' conclusion as well as your own>

Notes:

- You do not have to report the literature review from the academic paper, except possibly as a short summary stating if the bulk of the referenced articles are journals, websites or internal reports.
- You should write the report as you would do for reporting to your peers in either industry or academia. The document would be expected to be filed with summaries of other academic papers to allow researchers to peruse current research quickly.
- Use 11 point font, Times New Roman, 1.5 line spacing, no footnotes.
- This assignment is worth 12% overall; 4% for the first report and 8% for the second.

North Haugh, St Andrews, Fife KY16 9SS, Scotland
Direct Line: (01334) 463253 Switchboard: (01334)476161 ext 3253 Fax: (01334) 463278 e-mail: …@dcs.st-and.ac.uk

The first report should be chosen from list A, the second from list B. The same paper cannot be reported from both List A and List B. All papers are online from the ACM Digital Library, accessible from the electronic journals section of the University Library webpage.

List A (2 page report):
Choose one of the following:

1. '*A Class and Method Taxonomy for Object Oriented Programs*' David A Workman, ACM SIGSOFT Software Engineering Notes, p53-58,
   Vol 27(2), March 2002

2. '*Testability, Fault Size and the Domain-to-Range Ratio: An Eternal Triangle'* Woodward. M & Al-Khanjari, International Symposium of Software Testing and Analysis, ISSTA 2000, ACM SIGSOFT Software Engineering Notes, p168-172, Vol 25(5), 1-58113-226-2,

3. '*A Knowledge-Based COTS-Aware Requirements Engineering Approach'* Chung, L & Cooper, K., Procs. Of the 14th Int. Conference on Software Engineering and Knowledge Engineering, 2002 (SEKE 02), p 175-182, ACM 1-58113-556-4

4. '*Organisational Trails through Software Systems'* Knight, C & Munro, M, Procs of the 4th Int. Workshop on Principles of Software Evolution, Sept 2001,(IWPSE 2001), p 150-153, ACM (2002) 1-58113-508-4

List B (3 or 4 page report):
Choose any paper from the following conferences. The paper must be at least 6 pages long and preferably less than 15.

1. *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, 2002, Ischia Italy. (ACM International Conference Proceeding Series). Particular attention is drawn to the sessions on Requirements Engineering, Object-Oriented Technology, Measurement and Empirical Software Engineering and Reverse Engineering

2. *Proceedings of the International Conference on the Future of Software Engineering 2000*, Limerick, Ireland.
   All papers are roadmaps.

3. *Proceedings of the 22nd International Conference on Software Engineering 2000,* Limerick, Ireland, 2000 (ACM International Conference Proceedings)
   Particular attention is drawn to the papers by Subramaniam, Briand and Leszak

4. *Proceedings of the 24th International Conference on Software Engineering 2002, Orlando, Florida. (ICSE 2002)* (ACM International Conference Proceedings)
   Particular attention is drawn to the sessions on Empirical Methods, Requirements Engineering, Software Testing, Software Evaluation, Software Maintenance and Program Analysis.

CS3101 Databases

Practical Part 1 and 3

CS3101 Databases Practical

Part 1

Due: 7 March 2003 Credit: 40%

Aim
The aim of part 1 of this practical is to promote awareness of issues involved in designing databases.

Requirements
Specify in detail a scenario to be modelled. Design an E-R model to represent the data from the scenario. Translate the E-R model into a relational schema.

More details are given in the next page.

Submission
All work must be submitted electronically via MMS. No printed copies of any part are required. Contact dharini@dcs.st-and.ac.uk if you experience any problems with electronic submission.

School Policy
You are reminded of the School policy on academic fraud. Discussing practical work with others is perfectly acceptable; submitting another's work is not and will be treated as a disciplinary issue.

Dharini Balasubramaniam
17 February 2003

Details

Part 1 of the practical involves identifying and describing in English a real-world scenario to be modelled in a relational database, specifying what data the data model should contain, designing an E-R model to represent this data and finally converting the model into a relational schema.

Scenario
Some examples of suitable scenarios are given below. You are free to choose others.

- a banking system
- an airline reservation system
- a lending library
- a golf club
- a university

Clearly you will need highly complex models to capture these scenarios in their entirety. They will have to be simplified for the purpose of this practical. An example specification is given at the end as a guide to the level of detail required.

Tasks
Choose a scenario that you would like to model. Write a specification in plain English (instead of using a formal notation) of the data from the scenario that will be stored in the relational database. At this point, you should concentrate on what data should be stored rather than how it will be stored. Describe also the relationships and constraints between various parts of the data. Since this specification will be the basis for your database definition, it should be as precise as possible.

Design a representation of the chosen data in terms of entities, attributes and relationships. Construct an E-R model to depict this representation. Include cardinality constraints where applicable. As specified earlier, everything including the E-R model should be submitted electronically. It is recommended that you use a drawing tool but scanned versions of hand-drawn diagrams are also acceptable provided they are neat and legible.

Translate the E-R model into a corresponding relational schema. It should identify primary and foreign keys and null value constraints.

Documents to be handed in

1. A specification in English of the scenario to be modelled
2. An E-R model representing the scenario
3. A relational schema derived from the E-R model

© University of St Andrews

Example Specification of Scenario

Specification of a personal finance database

The database stores information about a number of personal accounts for a single person. The accounts can be either bank accounts or credit card accounts. Each account has an account number, current balance and issuing institution. Each institution has an address and a manager. A credit card account has a credit limit. A bank account has an interest rate and can be either a current account, which has an overdraft limit, or a savings account.

A number of associated transactions are stored for each account, recording for each one the payee, the amount and the date. A transaction may be in the past, in which case the database records whether or not it has been verified against an account statement. Alternatively it may be a future scheduled transaction, in which case it has a repeat interval specifying how frequently the transaction occurs.

Each transaction is classified as belonging to a particular category, e.g. salary, bills, groceries or car maintenance. Each category has an associated tax code.

It is required to be able to enter new categories into the database independently of transactions, i.e. it should be possible to store a category for which no corresponding transactions exist.

© University of St Andrews

CS3101 Databases Practical

Part 3

Due: 6 May 2003 Credit: 30%

Aim
The aim of part 3 of this practical is to promote awareness of database concepts through an implementation exercise.

Requirements
Create a Java program to capture the concept of relations including enforcement of uniqueness of primary key attributes and implementing the natural join operation for any two relations.

More details are given in the next page.

Submission
All work must be submitted electronically via MMS. No printed copies of any part are required. Contact dharini@dcs.st-and.ac.uk if you experience any problems with electronic submission.

School Policy
You are reminded of the School policy on academic fraud. Discussing practical work with others is perfectly acceptable; submitting another's work is not and will be treated as a disciplinary issue.

Dharini Balasubramaniam
27 March 2003

Details

Part 3 of the practical involves producing some source code in Java to model relations, operations and constraints.

It is important that your code is tested to ensure it is working as expected and well commented to indicate your reasoning.

Tasks

Define class(es) to model the concept of relations. These should take into account attributes, attribute types, schema and tuples (rows). An instance of *Relation* class should contain 0 or more rows of values for its schema. Primary key attributes should be specified and their uniqueness should be enforced. For the purpose of this practical, assume all attributes are of type string. It should also be possible to print out all rows of a relation instance.

Create an instance each for two relations with one or more common attributes (such as relations customer and depositor used in lectures). These should contain at least 3 rows of data each.

Print out all rows of the two relation instances.

Based on your class definitions, implement the natural join operation for two relations.

Apply the natural join operation to your relation instances created earlier and print out the rows from the resulting relation.

Documents to be handed in

Complete Java source including

1. Set of class definitions modelling relations
2. Instances of two relations
3. Implementation of natural join of two relations from 1.
4. Application of natural join operation relation instances from 2.

CS3102 Data Communications and Networks

Practical 1

CS3102 Data Comms and Networks, Practical 1
Due date: Friday 7th November *Note: You can hand it in earlier!!*
Value: 20

Hand in to mms.dcs.st-and.ac.uk in plain text, Word, html, or pdf *with appropriate suffixes*.

1 Describe the capabilities, protocols and standards implemented in the Nokia N-Gage. Give a critical review of this device. Are there any other protocols it could usefully implement? (2)

2 Shannon's Law suggests that a modem communicating across a telephone line with a S/N of 30db will be limited to a data rate of about 30Kb/s. The V.90 and V.92 modem standards allow for data rates in excess of 48Kb/s. Explain concisely how the ~30Kb/s limit is calculated, and why V.90 series modems can go faster. Describe the state-of-the-art V90 series modem characteristics. (2)

3 What motivated the designs of the i) Firewire and ii) USB communication standards? Draw a table showing the difference in capabilities and characteristics between Firewire 1.x, Firewire 2.x, USB 1.x and USB 2.x. Are there application areas where you would clearly choose either the most recent Firewire or the most recent USB interfaces? Explain. (2)

4 a) A 4Kb/s channel has a propagation delay of 20ms. For what range of frame sizes does stop-and-wait give an efficiency of at least 50% ? (1)

b) Suppose you design a sliding window protocol of a 1Mb/s point-to-point link to the moon, which has a one way latency of 1.25 seconds. What is the minimum number of bits you need for a sequence number if frames are fixed at 1 Kbyte? (1)

5 What are the purposes of the DSL and ASDL standards? Illustrate with some example applications. Under what circumstances would ADSL be used in preference to DSL? (2)

6 The full-duplex, fibre-optic link between St Andrews and Dundee is 20km long with a bandwidth of 155Mb/s (OC3). Assuming a negligible bit-error rate, 1 Kbyte frames, and a go-back-N sliding window protocol, how many bits are needed in the identifier field for efficient use of the link? Show your calculations. (2)

7 Draw a table summarising key features of the 802.11 wireless standards, 802.11a, 802.11b, and 802.11g. Include carrier frequencies, bandwidths, transmission distances, modulation and coding techniques, and shared medium access control methods. A comments column at the end of each row should identify one or more weaknesses of each approach. Provide a key for any acronyms used. (2)

8 The data "az" is to be carried by an HDLC I-frame as two 7-bit ASCII characters (no parity bits, therefore a total of 14 bits). Show the complete bit pattern of the frame, including delimiters, as it appears on the wire. To simplify CRC calculation use the CRC-8 generator, and an 8-bit FCS field. Assume a window size of 7. Label the fields clearly and use a fixed width font e.g. Courier or Monaco. (2)

Discretionary marks for well researched and/or presented answers. (4)

CS3103 Graphs and Algorithms

Practical 1

CS3103 Graphs and Algorithms Nov 2003, Practical 1

James McKinna

November 21, 2003 (minor revisions December 1, 2003)

This practical is due for submission on Fri. December 5th 2003 at MIDNIGHT. Late submissions will be subject to the usual penalties.

This practical is marked out of 50, and makes up 40% of the continuous assessment of the module, 8% of the overall assessment.

This practical is designed to:

- reinforce and test your understanding of the core concepts of the first part of the module: asymptotic complexity and the big-Oh notation.
- exercise your understanding of the main approaches to algorithmic problem solving considered so far: divide-and-conquer, greedy algorithms and dynamic programming

Submission should be via MMS in the form of a single .tar or .zip archive.

Documents should be readable in Word format or Adobe PDF.

Programming solutions should be submitted as `.java` source files, together with sufficient documentation to demonstrate the workings of your program. Please do NOT hand in class files or entire Together projects.

Algorithms should as far as possible be specified in pseudo-code, following the style adopted in notes—the principal consideration being that you should make clear how and why it is that your solution behaves as specified, and with the indicated (time) complexity.

Answer all the questions below

1. Consider the following four Java methods

```
public static int A(int n) {
        a = 0;
        for (int i = 0; i < n; i++) {
                System.out.println("hello\n");
                a++;
            }
        return a;
}

public static int B(int n) {
        b = 0;
        for (int i = 0; i < n; i++)
                b += A(i);
        return b;
}

public static int C(int n) {
        c = 0;
        for (int i = 0; i < A(n); i++)
                c += B(i);
        return c;
}

public static int D(int n) {
        d = 0;
        for (int i = 0; i < B(n); i++)
                d += C(i);
        return d;
}
```

For each method M, with running time $T_M(n)$ as a function of input n, find a simple function $m(n)$ of n such that $T_M(n)$ is $\Theta(m(n))$, i.e. $T_M(n)$ is $O(m(n))$ and $m(n)$ is $O(T_M(n))$. [10 marks]

2. Using the Master Theorem, or otherwise, calculate the running times for each of the following divide and conquer schemes:

   (a) Divide problem of size n into 5 problems of size $n/3$, with combination of solutions $O(n^2 \log n)$

   (b) Divide problem of size n into 3 problems of size $n/3$, with combination of solutions $O(n \log n)$

(c) Divide problem of size n into 1 problems of size $n/3$, with combination of solutions $O(\log n)$

(d) The obvious ternary variation on binary search: Divide problem of size n into *at most* 2 problems of size $n/3$, with combination of solutions $O(1)$

[10 marks]

3. Write your own Java implementation of the $O(mn)$ dynamic programming solution to the longest common subsequence (LCS) problem, detailed in the notes from before Reading Week. That is, write to the following interface:

```
public interface LCS
{
   public static int LCS(String s, String t);
}
```

using a class which iteratively computes the matrix $L(i, j)$. Supply enough documentation and test data to support your solution, in particular the overall running time of $O(mn)$.

[20 marks]

Marking Scheme:

Basic operation of program and testing	10
Quality of Design	3
Report	5
Bonus	2

4. Give, in pseudocode, an algorithm for outputting the longest common subsequence of two strings, in terms of the two strings and the data structures of your implementation in part 3.

That is, write to the following interface:

```
Algorithm: printLCS(s,t)
        Input: Two strings s, t (of lengths m, respectively n)
        Output: A string of length LCS(s,t) which is
                a common subsequence of the two strings.
```

Indicate, with reasons, the running time for your procedure in terms of m and n, and demonstrate its functional behaviour on some of the test cases which you have previously considered in part 3.

[10 marks]

CS3104 Operating Systems

Practical

CS3104 Practical

Aims and Objectives

The aim of this practical is to promote awareness of the issues involved in scheduling concurrent processes in an Operating System. The objective is to implement a number of simulated scheduling algorithms in Java and evaluate their performance.

Resources

A Java package *SchedulingSimulation.zip*, containing a working framework for the simulation, can be downloaded from:

`http://resource.dcs.st-and.ac.uk/Teaching/Modules/CS3104-OS/Practicals/`

The public API for the package is described at:

`http://resource.dcs.st-and.ac.uk/Teaching/Modules/CS3104-OS/Practicals/API/`

Detailed notes on completing this practical are given in the *Notes* section of this document.

Requirements

The practical is in 3 parts:

Part 1 Due: Mon W5
Implement a *First Come First Served* scheduling algorithm with multiprogramming, and test it on the supplied process workloads *A* and *B*.
To hand in: your algorithm `FCFSScheduler.java` and sample test output: verbose from *Workload A* and in summary format from *Workload B (credit 25%)*

Part 2 Due: Mon W7
Implement *Shortest Job First* and *Time Slicing* scheduling algorithms and test them on the supplied process workloads *A* and *B*.
To hand in: your algorithms `SJFScheduler.java` and `TimeSliceScheduler.java`, and sample test output from both as for Part 1 *(credit 50%)*

Part 3 Due: Mon W9
Test all three algorithms from Parts 1 and 2, and the supplied *Run To Completion* algorithm, on the supplied process workloads *C* and *D*. From the various outputs, deduce the nature of the two workloads.
To hand in: test output from the algorithms in summary format *(credit 5%)* and a report explaining your deductions from the output *(credit 20%)*

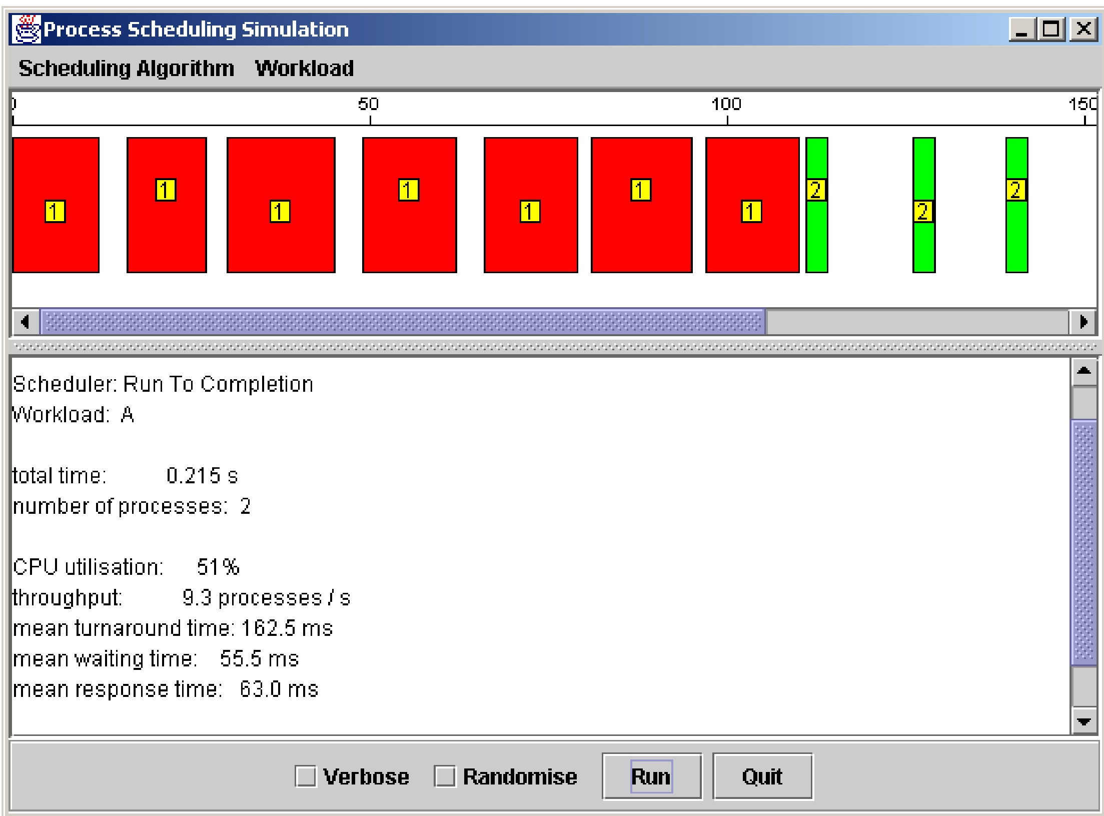
Algorithms, test output and reports should all be submitted in printed form. Source code for algorithms (the scheduler class definitions only) should also be uploaded to MMS. Additional credit will be given for well structured and commented code.

Notes

Running the Simulator

To get started, download the file *simulator.tar* from the web page and `unzip` or `tar xf` it to create a directory *simulator*. In Together, select *Open project*, browse for this directory and open *simulator/simulator.tpr*.

This should display an empty window. Click on the *Run* button to run a simulation using the smallest workload A and the provided scheduling algorithm, *Run To Completion*. The Gantt chart in the top pane shows the scheduling sequence, with time in *ms* labelled across the top. Each block, labelled with the corresponding process number, indicates a period during which that process has control of the CPU. Empty regions indicate periods during which the CPU is idle. An example is shown below:



The lower text pane contains a summary of various statistics. Selecting the *Verbose* option includes details of every process state change. This is useful while debugging algorithms, but will produce very lengthy output for any of the larger workloads (*B*, *C* and *D*). The *Randomise* option introduces a little random variation in process behaviour between runs; without it, successive runs will produce identical output. Even with the *Randomise* option selected, the balance of CPU and IO bound processes remains the same for any particular workload.

The *Workload* menu allows different process workloads to be selected:

- *A*: very short, containing one CPU bound process and one IO bound. Useful for algorithm testing.
- *B*: a longer sequence of 100 processes, starting at varied times, with equal mix of CPU and IO bound processes.
- *C* and *D*: sequences of 1000 processes, the nature of which is to be deduced in Part 3 of the practical.

The *Scheduling Algorithm* menu allows different algorithms to be selected. If a selected algorithm has not yet been implemented, an error message will be printed when it is run.

Implementation: Simulator Structure

The simulator knows about a number of scheduling algorithms, defined in the files `RunToCompletionScheduler.java`, `FCFSScheduler.java`, `SJFScheduler.java` and `TimeSliceScheduler.java`. In the downloaded version only `RunToCompletion.java` contains all the necessary code; the others are simply templates with empty methods ready to be filled in.

When the *Run* button is pressed, the simulator creates a number of *Process* objects, with the characteristics defined by the current workload. It then runs the simulation by incrementing the time repeatedly until all the processes in the workload have completed. As the simulation progresses and various events occur, the simulator calls the appropriate methods of the process objects. Events include: process arrival and termination; processes blocking for IO; interrupts indicating completion of IO requests; and regular timer interrupts. Each scheduler method defines the updates to process states and data structures to be made when the corresponding method occurs. In some scheduling algorithms, some of the events may be ignored.

The flow of control is thus directed by the simulator; your task is simply to implement the scheduler methods that will be called automatically by the simulator from time to time.

Processes

In order to be able to organise the appropriate execution of the processes, the scheduler needs to be able to discover the current state of a process, and to change it to a different state. The class *Process* provides methods for this as follows:

getState: This returns the current state of the process. It will be one of RUNNING, READY, BLOCKED or TERMINATED, represented as an integer as defined in class *State*.

setState: This sets a new state for the process. The parameter is one of the states described above.

Some process state transitions are made automatically by the simulator:

RUNNING to BLOCKED (when a CPU burst ends)
BLOCKED to READY (when an IO burst ends due to IO request being completed)
RUNNING to TERMINATED (when a process finishes)

Your scheduling algorithm need only initiate the following process state transitions:

READY to RUNNING (when a process is picked to run)
RUNNING to READY (when a process is pre-empted)

Note that you may assume that a process always starts and ends with a CPU burst. Thus its initial state is always READY, and there is never a transition from BLOCKED to TERMINATED.

The *Process* class also provides the following method, needed only by the *Shortest Job First* scheduler:

remainingBurstTime: This returns the time remaining during the current CPU burst for a RUNNING process, or the duration of the next CPU burst for a READY process.

Scheduler Structure

Each scheduler class provides the following methods/constructors, to be filled in for the new schedulers:

constructor:
The constructor initialises any data structures to be used by the scheduling algorithm. In the *Run To Completion* scheduler the constructor creates a new *First-In First-Out*

queue (see note below on data structures) to hold the processes in the order in which they are submitted.

processEntered (Process p):
This method is called by the simulator whenever a new process enters the system; the scheduler needs to add it to the appropriate data structure. In the *Run To Completion* scheduler the new process is run immediately if there is no process currently active, otherwise it is added to the end of the FIFO ready queue.

processBlocked (Process p):
This method is called by the simulator whenever the running process blocks due to an IO request. In the *Run To Completion* scheduler this event is ignored, since the current process retains control even during IO bursts.

ioInterrupt (Process p):
This method is called by the simulator whenever a blocked process becomes ready due to completion of its outstanding IO request. In the *Run To Completion* scheduler the process is immediately run, since only only the current process can issue IO requests.

timerInterrupt():
This method is called by the simulator at regular intervals. In the *Run To Completion* scheduler this event is ignored, since there is no pre-emption.

processTerminated (Process p):
This method is called by the simulator whenever the running process terminates. In the *Run To Completion* scheduler the next process at the head of the ready queue is removed from the queue and set running.

processesRemaining():
This method is called by the simulator to determine whether there are currently any processes in the system. A **false** result does not necessarily mean that the simulation run is finished, since it is possible for further processes to be added later. In the *Run To Completion* scheduler the method returns **true** if a process is currently running, or if the FIFO queue has non-zero length.

New Scheduling Algorithms

To implement the new scheduling algorithms, you need to edit the files `FCFSScheduler.java`, `SJFScheduler.java` and `TimeSliceScheduler.java` respectively. First, study the example scheduler provided, in `RunToCompletionScheduler.java`, to see how a very simple scheduler works. Assume a single processor in all of your scheduling algorithms.

Your *First Come First Served* scheduler will be broadly similar to *Run To Completion*, but it needs to ensure that when the running process becomes blocked, another ready process is allowed to run immediately. When the blocked process becomes ready at the end of its IO burst, it should be put to the back of the queue.

The *Shortest Job First* scheduler should be pre-emptive. At every opportunity it should determine whether any other process would be able to complete a CPU burst faster than the currently running process. If so, that other process should be allowed to run immediately. This is made possible by the (highly artificial) ability to ask a running process how long its current burst will last, and a ready process how long its next CPU burst will be (see notes in *Processes* section earlier).

The *Time Slicing* scheduler should also be pre-emptive: after a given period the currently running process should be pre-empted and another process given a turn on the CPU.

Once you have written the details of a particular scheduler, rebuild and rerun the simulator using the same commands as before. It may be helpful to enable the *Verbose* option to print a trace of each process state change; but this will be extremely slow on any workloads other than *A*.

Data Structures

Most scheduling algorithms need to keep track of ordered queues of processes, and/or unordered sets. You may find the classes `java.util.Vector` and `java.util.HashSet` useful for these. The most relevant methods are `add`, `remove`, `elementAt` and `size` for `Vector`, and `add`, `remove` and `size` for `HashSet`. The file `RunToCompletion.java` contains examples of `Vector` use.

Full details of the APIs provided by these and other standard classes can be found at:

`http://resource.dcs.st-and.ac.uk/Modules/CS2001-ADS/java/jdk1.2.2/`

Output for Handing In

In all three parts of the practical you are asked to hand in test output from the scheduling algorithms. To avoid this becoming too lengthy, you should collect verbose output from workload A, and the statistical summary only from other workloads. Although it is not obvious, you can copy text from the text output pane by selecting it and typing *Ctrl-C*, after which it can be pasted into another document.

Plagiarism

You are reminded of the School policy on plagiarism: discussion of practical work with others is perfectly acceptable. Submission of another s work is not, and will be treated as a disciplinary issue.

MJL/AR October 2002 (after GK 2000)

CS4101 Artificial Intelligence

Practical 3

CS4101 Practical 3

Constraint Satisfaction Problem Solver

For this practical you must write code to solve random CSPs.

You will be provided with a Java class that calls the *uniform random binary CSP* generator developed by Daniel Frost, Christian Bessiere, Rina Dechter and Jean-Charles Regin (http://www.lirmm.fr/~bessiere/urbcsp.c). This C program must be compiled like so:

```
[user@host]$ gcc –o urbcsp urbcsp.c
```

and the `urbcsp` binary must be left in the same directory as the Java classes. Consult the website http://www.lirmm.fr/~bessiere/generator.html for details of the parameters that this generator takes (See the section entitled "The random problem model").

To aid your progress, the provided Java classes take the parameters mentioned above (except for the number of instances), generates the relevant CSP and parses the resulting file that contains the CSP. Skeleton `Variable` and `Constraint` classes have been written that store all the relevant information in the generated CSP. Your task is to extend or modify these classes to add functionality that will allow you to solve the CSP.

This is to be done by traversing the search space of the problem to find a solution. The best way to do this is to write a simple backtracking algorithm that traverses the tree by performing a *Depth First Search*. At every node in search ensure that none of the constraints have been violated and/or none of the `Variable` objects have empty domains. If one of the constraints has been violated (or a variable has an empty domain) backtrack from your previous choice and remove it from the domain of that variable.

The output should look **exactly** as below. If a solution is found:

```
Solution Found:
      V0 = 0
      V1 = 3
      V2 = 4
      V3 = 2
      V4 = 1
Backtracks: 132
Runtime: 24.3 seconds
```

or like this if there is no solution:

```
No Solution Found
Backtracks: 429
Runtime: 64.207 seconds
```

There will be a small prize of 2 bonus marks to the student with the fastest constraint solver (with a maximum of 20/20 over the three practicals). For those interested in making their solver more efficient they may wish to look at *dynamic variable ordering heuristics* and increased levels of *consistency* e.g. forward checking, arc consistency.

Finally, you should write a small report detailing your implementation and any optimisations you made or difficulties you faced.

The deadline for this essay is 5th December.

What to hand-in: Submit code and report via MMS. The documents must be zipped into a single archive that contains just the report and source code only i.e. no class files or IDE files.

Marking Guidelines:

Backtracking algorithm	2
Constraint verification	2
Correctly formatted output	1
Report	1
Good coding practices and standards	1
Discretionary	1

CS4102 Computer Graphics

Practical 2

CS3034 Computer Graphics Practical 2

Recursive ray-tracing

Bernard Tiddeman 27th March 2002

Introduction

In this practical you are asked to implement a recursive ray-tracing program as a Java[tm] 1.0 applet. You will be given a number of helper classes to get you started. These contain concrete and abstract base classes which support:

a) calculating the surface shading.

b) calculating the reflected and refracted rays at each intersection.

c) wrapping and sampling textures.

d) interpolating a number of values across 3D triangles.

Your task is to implement concrete sub-classes for various objects, lights and surface effects. Efficiency is not an important consideration (ray-tracing is a slow method for all but the simplest scenes) and there are no marks for the user interface. See the marking scheme below for details.

Marking scheme

Type	*Objects/Features*	*Marks Available*
Applet/program	Set up viewing vectors	1
	Render method	2
	Recursive ray-trace method	2
Objects	Polygon mesh (flat shading)	4
	Polygon mesh (interpolated vertex shading)	4
	1 other shape (e.g. sphere, torus, cylinder etc)	1
Lights	Directional and point coloured lights (no shadows)	3
	Lights which cast shadows (no transparency)	3
	1 of attenuated, cone or flap lights	1
Surface effects	Texture mapping	4
Total	**10% of module**	**25**

Please note, you only have to give ***one example*** of each object or feature e.g. only one shape needs to be texture mapped, or only one type of light needs to casts shadows to gain the marks.

Files provided

Vector3D, Matrix3D, RayTraceObject, RayTraceLight, RayTraceMaterial, RayTraceTexture, RayTraceWrapper, RayTraceCylinderWrap, RayTraceFlatWrap and RayTraceTriangleInterpolator classes are provided in the *classes* directory. Look in the *javadoc* directory for more details.

Some *images* and *meshes* are available in the appropriate directories.

Hints and tips

Get something simple working first e.g. a sphere or triangle with no texture mapping and a directional light with no shadows.

Texture mapping a sphere using the cylindrical wrap is easier than reading the mesh files.

Only work on the parsing the mesh files after you've checked the rendering of simple polygons, e.g. triangles.

Polygons with more than 3 vertices can be split into triangles when loading for rendering.

For hit testing of polygons, you can use the java.awt.Polygon.inside(x,y) method. This is an integer method, so you might want to multiply the 2D coords by some large number to avoid problems.

The shadow tracing is non-recursive and can be put in the light's getLight method i.e. search through the object list for any intersections, checking only that its between the light and the surface.

Warning: Graphics programming can take up a lot of time, don't take up more time than its worth!

Mesh File Format

These polygon meshes are stored in the OFF format which are in the form:

OFF

<vertex count, n> <face count, m> <edge count, l>

<vertex 1>

<vertex 2>

...

<vertex n>

<face1>

<face 2>

...

<face m>

<edge 1>

<edge 2>

...

<edge l>

In this assignment we are ignoring edges. Vertices are simply 3 floats i.e.

x-coord y-coord z-coord

Faces are stored as the number of vertices in the face (n), followed by the index of each vertex in the vertex list (above) i.e.

$n\ v_1\ v_2\ \ldots\ v_n$.

Deadline

Friday 10th May 2002 (provisional).

CS4103 Distributed Systems

Practical 1

CS4103 Distributed Systems
Practical 1

This practical is worth **10** out of the 20 marks allocated to practical work.

It is **due** on *Tuesday 18 November 2003*.

Deliverables

Your submission should be generated electronically via some appropriate package (such as PowerPoint, as I used for the lecture notes: see the prototype proofs.ppt), and uploaded to the allocated MMS slot.

Background

In the lectures we have studied the correctness properties of some mutual exclusion (*mutex*) algorithms, both centralised and distributed. We constructed rigorous safety proofs for the Dekker-Peterson algorithm and the 2-process version of the Bakery algorithm.

Below you will find the Dekker-Peterson algorithm, the general (i.e. polyprocess) Bakery algorithm, and two other mutex algorithms: Dijkstra and Anon.

1. Find a counterexample to show that Anon is not safe (hence the lack of name!).

2. Using the notes as a guide, devise rigorous safety proofs for the Dijkstra algorithm and for the general Bakery algorithm.

   You should in fact be able to adapt the 2-process proofs very easily. Comment on why this is possible: is it the particular style of proof, or the nature of the problem?

3. If the Bakery algorithm really is distributed, the read of a remote variable — e.g. of n_1 by p_0 — is asynchronous, with the implication that the value seen by a reading process may no longer be the value in the variable. Any proof must must take account of this, and we argued that our proof of the 2-process version does, but relies on careful interpretation of the events appearing in it.

   Devise a more elaborate version of the 2-process proof in which there is a separate time axis for each process and message passing is made explicit.

4. Here is a statement about the Dijkstra algorithm:

   p_i can enter its critical section only if $req_j < 2$ for all $j \neq i$

   It appears in an operating systems textbook, where it is considered sufficently "obvious" to form a premiss of a safety proof of the algorithm — is this reasonable?

Algorithms

Dekker-Peterson (centralised, 2-process)

```
int req[0..1] = {0,0};
int turn = 0;
p[i=0..1]: repeat {
   req[i] = 1;
   turn = 1-i;
   while(req[1-i] & turn == 1-i);
   critical section...
   req[i] = 0;
   other stuff
}
```

Bakery (distributed, polyprocess)

```
int x[1..N] = {0,…,0}
int n[1..N] = {0,…,0}
p[i=1..N]: repeat {
   x[i] = 1; n[i] = max_{j≠i}(n[j]) + 1; x[i] = 0;
   for(j ≠ i) {
      while( x[j] );
      while( n[j] > 0 && n[j] « n[i] );
   }
   critical section...
   n[i] = 0;
   other stuff
}
```

Dijkstra (centralised, polyprocess)

```
int req[1..N] = {0,…,0};
int turn = 1;
p[i=1..N]: repeat {
   for( x = 0; x ≤ N; ) {
      req[i] = 1;
      while( turn ≠ i ) if( req[turn] == 0 ) turn = i;
      req[i] = 2;
      for( x = 1; u ≤ N && (x == i || req[x] < 2); x++ );
   }
   critical section...
   req[i] = 0;
   other stuff
}
```

Anon (distributed, 2-process)

```
int req[0..1] = {0,0}
p[i=0..1]: repeat {
   for( req[i] = !req[1-i]; req[1-i]; req[i] = !req[1-i] );
   critical section...
   req[i] = 0;
   other stuff
}
```

CS4302 Multimedia

Practical

Multimedia Practical: A basic presentation system

Aim: To gain familiarity writing multimedia delivery software using the java multimedia classes including graphics, text, sound and jmf (java media framework) classes..

Objectives: The objective of this practical is to write a presentation application (i.e. a "mini-PowerPoint") which can display a number of different slides with text, graphics, sound, animation and embedded video clips.

Specification

The presentation application should be able to display at least the following 4 types of slide:

1.**Title slide**, with centred title, author, and room for additional info.

1.**Bullet point slides**, with title and a variable number of bullet points.

1.**Image slides**, with a title, centred image and figure caption.

1.**Video slides**, with a title, centred video and a caption beneath the video.

It should also include at least the following 2 animation effects:

1.**Bullet point animation** (e.g. slide in from right).

1.**Slide transition** (e.g. fade-in).

It should also **play a midi sound** during either (or both) of these 2 animations.

All of the text and graphics drawn should be **anti-aliased**.

The animation should use **double-buffering** (i.e. rendering first to an off-screen image, then displaying the image on-screen) to produce smooth animation. The slide's contents (titles, bullet points, image file names etc) should be stored in a simple text format and parsed using the StreamTokenizer class.

Resources

Use the examples (SimplePlayerApplet and MyMidiPlayer) in the resource directory on-line, for examples of how to implement the video clip and the midi sound. You will also find some images and video clips to use in your presentations, and an example of the type of file that your application might read (presentation1.txt).

Assessment

Feature	**mark**
Title Slide	**1**
Image Slide	**1**
Bullet Slide	**5**
Slide Transition	**2**
Video Slide	**6**
Total	**15**

This part of the practical work is worth 15% of the module grade, the remaining 5% will come from the presentations in the second half of the course.

What to hand in

Fully (javadoc) commented source code and a 1-page explanation of your code. You should also arrange a time to demo your working program.

Deadline

Friday 29th November.

Bernard Tiddeman, September 2002