Imperial College of Science, Technology and Medicine	University of London
Computer Science (CS) / Software Engineering (SE)	BEng and MEng Examinations Part I
Department of Computing	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations
and is a required part of the degree assessment.
Laboratory work must be handed in for marking by the due date.
Late submissions may not be marked.

Exercise: 6	Working: Individual
Title: Recursion in Java	
Issue date: 17th November 2003	Due date: 24th November 2003
System: Linux	Language: Kenya/Java

Aim

- To provide an introduction to the programming language **Java** and the **Kenya** interactive program development environment that you will be using; this provides a simplified version of Java.

Introduction

- Java is an *Object Oriented* programming language and in term 2 you will be learning to write programs that use a fuller range of Java.

  In term 1 you will be taught about problem solving using basic programming constructs and so you will not need to learn all of the features of the Java language. In particular you will not be using its *Object Oriented* features.

  The simplified version of *Java* that you will be using is called *Kenya*. From now on we will talk about Kenya as if it were a distinct programming language although it is really a simplified form of Java.

  You first write Kenya programs and these are automatically translated into Java. The Java program is then compiled and executed. The process works as follows:

  1. You open the Kenya system and write and edit your Kenya program.
  2. You check the Kenya code and translate it into Java.
  3. You compile the Java code.
  4. You execute the compiled Java program that you have written.

- For the term 1 labs you only need to learn Kenya. You can program entirely in the Kenya programming environment. If you wish you can program directly in Java. When you submit your programs you *must* make sure that a Java version of your program is present. The Kenya version of your program is optional but you should submit a Kenya version if you developed your solution using Kenya.
- As the Kenya programming environment produces Java code you can study the Java translation of your program. You may find this a useful preparation for term 2.
- Further details about Kenya can be found in the "Kenya User Guide". The URL for this is *http://www.doc.ic.ac.uk/lab/kenya/help/* or you can follow the link from the Comp 1 web page.

The Problem

- You should write four programs in Kenya/Java.
- **Fibonacci.k/Fibonacci.java** which computes a term of the Fibonacci series.
- **eApprox.k/eApprox.java** which calculates an approximation to e.
- **Hanoi.k/Hanoi.java** the Towers of Hanoi puzzle.
- **PrintBase.k/PrintBase.java** which prints a non-negative integer in a given base.

Submit by Monday 24th November 2003

Using Kenya

- When you are developing programs you will be spending much of your time viewing, editing, running, debugging and re-running your code. The **Kenya** programming environment provides an interactive environment for you to do this.

  An introduction to **Kenya** is given in the "Kenya User Guide", however, for now it will probably be enough to step through the simple viewing, editing and running sequences given below.

- Enter **Kenya** by typing:

  ```
  kenya &
  ```

  at your linux prompt. After some time you will be given a **"Kenya"** window. Type this simple Kenya program in the Kenya window:

  ```
  void main()  {
    println("Hello world!") ;
  }
  ```

- You will now need to check the program for errors and translate it into Java. Click on the **"Translate"** button on the tool bar. The first time that you do this you will be prompted for a file name to store the program. Type **Hello.k** or **Hello**. In both cases your program will be stored in a file called **Hello.k**. Once this is done, the program will be translated into Java. You should get a message printed in the Message window below your program telling you:

  ```
  Finished parsing
  ```

  Your Kenya program should now be translated into Java. (You don't need to concern yourself with the Java code but if you want to look at it you can click on the button marked **"Java"** above the code window. The buttons marked **"Java"** and **"Kenya"** allow you to toggle between Java and Kenya.)

  If you have a syntax error these will be described in the Message window. Kenya should also highlight the place in the code where the error was found. If there is an error you should correct the error and then select **"Translate"** again.

- Now you have translated your Kenya program you will need to compile the Java code by clicking on the **"Compile"** button next to the **"Translate"** button. This may take some time but hopefully you should see the message

  ```
  Compiled OK
  ```

- You can now run your program using the **"Execute"** button. You will then see the message

  ```
  Running ...
  ```

  and a **Program Output** window should appear containing the program output

  ```
  Hello world
  ```

- You can save your program to a file by selecting the **"Save"** button. You should save your program periodically as you develop it. Please note that files containing Kenya code should have a **.k** extension, so your file should be called something like **Hello.k**.

- When you have finished developing your program you can quit the Kenya environment by clicking the **"Quit"** button.

- Another way to start **Kenya** is by giving the name of the file containing the **Kenya** program as an argument with the command

  ```
  kenya Hello.k &
  ```

- This should serve as a simple introduction ot the **Kenya** environment. Further details about it e.g. editing commands etc. are given in the "Kenya User Guide".

The programs you should write

Fibonacci.k/Fibonacci.java

- Write a Kenya/Java program **Fibonacci.k/Fibonacci.java** which prompts the user for a non-negative integer to be used as a *position* in a **Fibonacci series**, followed by two further integers to be used as the *starting values* of the series (positions 0 and 1 respectively). The program should then output the Fibonacci number at the specified position in the series. This is a similar problem definition to your earlier Haskell exercise. As a reminder: A Fibonacci series starts with two given values, and each subsequent term is the sum of the previous two terms. For example if the starting values are 2 and 7, the series would be: 2 7 9 16 25 41 66 107 .....
  We consider the *positions* to be numbered from zero onwards (0,1,2,3,...), so for example position 5 in the example series above contains 41.

- Your program should print an appropriate prompt string or strings before reading in the input values, and an announcement string before writing out the answer. To help with autotesting, the answer should be the **last** thing output. Also, make sure you read in the input values in the order given above!

- The computation should be done by a recursive method separate from the main program, called for example

  ```
  int fibonacci( int n, int first, int second )
  ```

  The main program part will then deal solely with I/O and a single call to this method.

eApprox.k/eApprox.java

- The constant e is defined using the following expansion:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...$$

  Write a program that prompts the user for a non-negative integer that gives the number of expansion steps required to produce an approximation of e. For example the **0**'th expansion of the series is `1`, the **1**'st expansion is `2`, the **2**'nd `2.5`,.. etc. Your program should print a suitable message reporting the approximation to e as calculated such as:

  ```
  The approximation of e is 2.5
  ```

  Again, to help with autotesting, the answer should be the **last** thing output.

Hanoi.k/Hanoi.java

- There is an ancient legend that in Hanoi there is an elegantly-crafted cosmic puzzle which controls the birth and death of the world. It consists of three **pegs**, which we will call 'A', 'B' and 'C', and (initially, at the beginning of time) a **tower** of 64 **disks**

with holes in the middle, stacked up on **peg A.** The disks are all different sizes, and are stacked in order of size, largest at the bottom, smallest at the top. A team of untiring priests works endlessly in shifts, moving one disk at a time, from one peg to another, with the aim of transferring the entire tower from its original **peg A** to **peg C**. But they must at all times obey the sacred rule, which is that a larger disk can never be placed on top of a smaller disk. When they complete their task time will end and the world will crumble to dust. Fortunately finishing the puzzle will take a long time.

- Try the Towers of Hanoi puzzle with some coins or scraps of paper to represent the disks. There can be any number of disks, but there must be exactly three pegs. Four disks is a good number to practise with.

- Write a Java program **Hanoi.k/Hanoi.java** which prompts the user for a non-negative integer, and then solves the Towers of Hanoi puzzle recursively, using the algorithm given below. For example, if there are two disks, your program should print:

```
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
```

  If there are zero disks to be moved, **do nothing**. To move N > 0 disks from peg **from** to peg **to** via "spare" peg **via**, then:

  - **move n-1 disks** from **from** to **via**, with **to** playing the role of spare;
  - **move disk n** from **from** to **to**;
  - **move n-1 disks** from **via** to **to**, with **from** playing the role of spare.

PrintBase.k/PrintBase.java

- Write a Java program **PrintBase.k/PrintBase.java** which prompts the user for a non-negative integer, then prompts for a base to express it in, and then prints out the given integer in the given base by calling a recursive helper method.

- Your program should cope with bases from 2 to 16. You should follow the convention that the digit values from 10 to 15 (base ten) are printed using the letters A, B, C, D, E, F. You will need to write a method **putOneDigit** which prints a single digit value (0..9,A,B,C,D,E,F) accordingly.

- As with **Fibonacci.k/Fibonacci.java**, your program should print suitable prompt and announcement messages to make its interactive usage simple and natural. Again, to help with autotesting, the answer should be the **last** thing output.

Submission

- Before submitting, you should **test your programs on a wide range of inputs**.

- The automatic submission system will look for two files for each program, a Kenya file and a Java file. The Kenya file should have a **.k** extension and the Java file should have a **.java** extension. The Kenya environment produces a Java file with the same name as the original file and you should check that this file is present before submission. If you have written the program in Java without using the Kenya environment the automatic submission system will collect the Java files but give a warning that the Kenya files are missing. You can if you want write the program in Java directly.
- Submit your programs **Fibonacci.k**, **Fibonacci.java**, **eApprox.k**, **eApprox.java**, **Hanoi.k** and **Hanoi.java**. **PrintBase.k**, **PrintBase.java**, by typing the command **submit 6** at your Unix prompt.

Assessment

Fibonacci	1
eApprox	1
Hanoi	1.5
PrintBase	1.5
Design, style, readability	5
Total	10