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: 13	Working: Individual
Title: Turtle Graphics	
Issue date: 9th February 2004	Due date: 16th February 2004
System: Linux	Language: Java

Aims

- To design and implement a non-trivial Java program that implements a simple *turtle graphics* application
- To implement a *command interpreter* that reads and executes commands typed in by a user
- To provide practice at using one of Java's predefined *collection* classes
- To introduce *iterators* for traversing a data structure

The Problem

This exercise requires that you design and write a complete Java program which will allow a user to draw pictures using *Turtle Graphics*. In the original implementation by Seymour Papert the "turtle" was a small robot which could be driven using simple commands like "turn left 60 degrees", "move forward 2 feet" etc. These could be composed together using a special purpose language called *LOGO*. The important feature of the system is that there is no global frame of reference, and all motion is relative to the current position and orientation of the turtle.

In your implementation you will use the same idea to drive a two-dimensional graph plotting device equipped with a pen. You will simulate this device by "drawing" lines of characters on the screen. You will represent the paper being drawn on using a two-dimensional array of characters and display the picture when the user issues the appropriate command.

- To write the program you should design and write whatever data structures and methods are appropriate. The detailed design of the program is left up to you, although the basic structure required is outlined below.
- Your program should read a sequence of turtle commands from the keyboard and terminate at the end of file indicated by typing *Control-d* at the keyboard on a line of its own. Alternatively, you can run your program on our test files (or additional ones of your own) with input redirection as described in the student sorting exercise.
- The idea is to implement different 'flavours' of turtle that behave in different ways when they reach the edge of the paper. A turtle may "freeze" at the edge waiting to be turned before moving. Alternatively, it may "bounce" and return across the paper, or it could "wrap around" and enter the paper on the opposite side. For this exercise you are asked for the assessed part to implement two different sorts of turtle, a turtle that freezes at the edge and a turtle that bounces by turning around by 180 degrees and retraces its steps. You can also implement other sorts of turtle, although these will not be assessed.
- Your program should allow the user to create any number of named new turtles of each type and issue commands to them individually.
- You are strongly advised to use the I/O methods provided by the class **kenya.io.InputReader** that you have used in the first term.

Submit by Monday 16th February 2004

What To Do

Copy the the skeleton program file **TurtleInterpreter.java** and the test data files (**turtle_test1.dat**, **turtle_island.dat**) and corresponding output results files (**turtle_test1.res**,**turtle_island.res**) provided using the command **exercise 13**. You can use the test files to test your program using *input redirection*. Once you have copied this file you should create a subdirectory **turtleSupport** for the package containing the support classes your program will need, **Turtle.java**, **BouncyTurtle.java** and **Paper.java**.

Driving the Turtle

To drive your turtle you will need to implement a simple language to specify pictures from the keyboard. **We shall use this language to auto-test your program** so you must implement the six commands with exactly the syntax given below.

Each command starts with a keyword (shown in bold) followed by a variable number of parameters (shown in italics). All keywords and parameters are separated by white space (blanks, tabs and newlines) and all numeric parameters are integers.

- **paper** *height width*
  Set the paper size to *height* lines deep and *width* characters wide (where *height* and *width* are integers). This command should set the area of the paper given to blanks and reset all the turtles. When a turtle is reset, the plotting character is set to "*", the pen

is raised from the paper and the turtle is placed at the top left hand corner of the paper facing **south**. This will always be the first command entered, but you should also be able to use the command any number of times subsequently in your program to start a new picture.

- **new** *type name ypos xpos*
  This creates a new turtle of initially a *bouncy* or *normal* type called *name* placed at position (*ypos*,*xpos*) (downwards and rightwards respectively, so that (0,0) would mean top left) on the paper, pen up, with plotting character "*" and facing **north**. The type should be given as the string **bouncy** or the String **normal** and the name of the turtle can be any String appropriate.

- **pen** *name state*
  where *state* is either **up** or **down** or a single *non-blank* character. **up** and **down** set the pen to the specified state, otherwise the plotting character is changed to the one specified. **Note:** you can find the character at a specified place in a *string* object by using the method **charAt(int n)**. The command is carried out by the turtle called *name*.

- **move** *name length*
  moves the pen *length* units in the current direction. The command is carried out by the turtle called *name*.

- **right** *name angle* and **left** *name angle*
  Respectively turn the turtle *angle* degrees clockwise or anti-clockwise (taking the parameter to the nearest 45 degrees). The command is carried out by the turtle called *name*.

- **show**
  Print the contents of the paper on the screen.

Your program should read and execute sequences of commands (possibly creating more than one picture) until end-of-file.

You should design the program using an appropriate class structure (for example the different sorts of turtle have a lot in common). Clearly you should look to building a **Turtle** superclass with a view to implementing specific subclasses for turtles that reflect, wrap, bounce etc. For the assessed part of this exercise you will need to write two classes **Turtle.java** and **BouncyTurtle.java** for the turtles that "freeze" or "bounce" respectively.
Some commands concern the paper and each turtle needs to be able to refer to the paper when making a move. This suggests the need for a separate class **Paper**

Many of the above commands only involve changes to the state of a Turtle (e.g. **pen**).

You will need to store your turtles in a data structure so that you can refer a command to the appropriate turtle. You can implement any structure you wish, but are encouraged here to explore one of the Java collection classes, such as **LinkedList**. This has a number of associated methods; for this exercise you only need to use the methods **void addLast(Object x)** that adds an object to the list, and **ListIterator listIterator()** that returns an

iterator over the list. To use a **ListIterator** here you simply need to use the two methods **boolean hasNext()** which returns true iff the list contains at least one element, and **Object next()** that returns the next object in the list. Note that calling **next()** has the effect of internally advancing the iterator to the next element in the list, if there is one.

Note that when you remove an object from a **LinkedList** of turtles the result will be an **Object**, i.e. the most general object type, even though you know it's going to be a **Turtle**! You therefore have to *cast* the **Object** into an **Turtle**, for example:

```
turtle = (Turtle)( it.next() ) ;
```

where `it` is the name of the iterator. Compare this with Haskell which supports proper polymorphic data types.[1]

Unassessed

- You can extend the turtle language using further commands that you design yourself. As we cannot autotest any additional commands you should explain them using comments and even demonstate them to your PPT.
- You can also design and create new types of turtle, such as a "wrap-around" turtle. Again, you should explain the new turtle

Testing Your Program

- Each class should be implemented and tested separately as far as possible. You should test your program on the various error conditions to ensure that your program behaves robustly.
- The supplied results files give illustrative output obtained from the test data files, although different valid interpretations of the meanings of the various turtle activities may of course give different results.

Submission

- You should submit *four* files:
- **TurtleInterpreter** that implements the command interpreter and the support classes **Turtle.java**, **BouncyTurtle.java** and **Paper.java**
- Submit these files by copying **TurtleInterpreter.java** into the **turtleSupport** subdirectory then *cd* into the **turtleSupport** subdirectory and type: **submit 13** at your Linux prompt.

[1] This *is* being fixed! A version of Java called GJ (Generic Java) essentially allows parameterised types in the same spirit as Haskell.

Assessment

Command line interpreter	1
Turtle movements	3
Displaying content of paper	1
Design, style, readability	5
Total	10