Imperial College of Science, Technology and Medicine	Department of Computing
Computing Science (CS) / Software Engineering (SE)	BEng and MEng
Examinations Part I	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: 12	Working: Individual
Title: Turtle Graphics	
Issue date: 9th February 2009	Due date: 16th February 2009
System: Linux	Language: Java

Aims

- To design and implement a non-trivial Java program that implements a simple *turtle graphics* application
- To provide further practice with packages in Java.
- To implement a *command interpreter* that reads and executes commands typed in by a user
- To use abstract classes in designing a program.
- To practice with *inheritance* and *overriding*.
- To gain experience with using *generics* and *collections*.

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. To do this you will need to create an input file (such as those provided) and then run your program:
  `java -ea TurtleInterpreter < turtle_bouncy_test.dat`.
- For this exercise you are asked to implement four different 'flavours' of turtle that behave in different ways when they reach the edge of the paper. A turtle may **continue** when it reaches the edge and go off into space - continuing to track its position and follow instructions - even if it wanders around in space forever and never returns to the paper. It may bounce and return across the paper. It could either **bounce back** by turning 180 degrees on or it could **reflect back** in the way a light beam would when it is deflected by a mirror (i.e. at the same angle it made with the perpendicular drawn from the edge of the paper). Alternatively, it could **wrap around** and enter the paper on the opposite side or corner.
- For this exercise you are asked to implement one basic turtle type **Turtle** dealing with the default behaviour of a generic turtle. This should be an *abstract* class with an abstract method to deal with the turtle's behaviour at the edge of the paper. The other types of turtle should implement this abstract method and use inheritance for any shared attributes and methods. You should define the **BouncyTurtle**, **ContinuousTurtle**, **WrappingTurtle** and **ReflectingTurtle** as subclasses of the **Turtle** superclass.
- Note for the purpose of this exercise the behaviour of a turtle is as follows: when it is instructed to move it moves to the next location and *then* places a mark at this location (provided the turtle is on the paper and has its pen down).
- Your program should allow the user to create any number of named new turtles of each type and issue commands to them individually using their names.
- Java 1.5 introduces a class **java.util.Scanner** which reads input from a variety of sources including File, String, or InputStream. Since the standard input, **System.in**, is a type of InputStream, you may use this class to retrieve input fed in from the console. The Scanner breaks up input into 'tokens', which can be converted to different types via different methods. e.g.:

```
Scanner sc = new Scanner(System.in);
while(sc.hasNext()){
  String word = sc.next();
  int number = sc.nextInt();
}
```

Refer to the Java-API for more information on this class.

To use these your class file **TurtleInterpreter.java** will need to contain the line: `import java.util.Scanner` ; at the beginning.

Submit by Monday 11th February 2009

What To Do

Copy the the skeleton program file **TurtleInterpreter.java** and the test data files (**turtle_bouncy_test.dat** .. etc.) and corresponding output results files (**turtle_bouncy_test.res** .. etc.) provided using the command **exercise 12**. You can use the test files to test your program using *input redirection*.

Once you have copied these files you should create a subdirectory **turtleSupport** for the package containing the support classes your program will need, **Turtle.java**, **BouncyTurtle.java**, **ContinuousTurtle.java**, **WrappingTurtle.java**, **ReflectingTurtle.java** and **Paper.java**. These should all be in *package turtleSupport*; whereas your main program class **TurtleInterpreter.java** is an importer and user of the services offered by this package and should be at top level, i.e. not in a package.

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.

This part of the program is called a *command line interpreter* and should be simple and straightforward to write.

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. This can be a *bouncy*, *continuous*, *wrapping* or *reflecting* type depending on the *String* supplied. The user must supply one of these 4 Strings as the

type, in lower case. The turtle is called *name* and all future commands are directed to the turtle using this *name*. The turtle is 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**.

- **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. Many of the commands only involve changes to the state of a turtle (e.g. **pen**). In this case you will need classes to represent the different types of turtle. You will need a superclass **Turtle** and some subclasses, as listed earlier, to implement the behaviour of the different sorts of turtle.

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**.

As you may have several Turtles on your paper at any time you will need to store your turtles in a data structure so that you can pass on a command to the appropriate turtle.

Although you could use an array to store your turtles you are recommended to store them using one of the Java collection classes, such as **LinkedList**. This class has a number of associated methods that allow you to add and remove items and perform other operations. The class also contains an *iterator* method that returns an iterator that allows you to step through the list.

If you use a **LinkedList** to store your turtles you only need to use the methods **void add(Object x)** that adds an object to the list, and **ListIterator listIterator()** that returns an *iterator* over the list. Java version 1.5 supports polymorphic data types in a way similar to Haskell so you will need to provide the type as a parameter when you instantiate the generic

class.

Since the **LinkedList** class implements the **List** interface you can declare a variable of type **List** and then assign a **LinkedList** to it. The code for declaring a linked list of turtles is:

```
public List <Turtle> turtles = new LinkedList <Turtle> () ;
```

This declares that the list *turtles* will be made up of objects of type *Turtle*. Similarly when you want to instantiate a **ListIterator** for searching the linked list you need to parameterise the declaration.

```
ListIterator <Turtle>  it = turtles.listIterator( 0 ) ;
```

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.

Designing the Turtle class

- When you design the Turtle you will need to consider what attributes a turtle needs as many of the commands will simply change the turtle state.
- You will need to think of how you are going to decompose the problem into specific turtle methods so that the program class can be extended simply. You should aim to write the superclass so that you can inherit most of its attributes and methods in the subclasses. Different turtles will certainly behave differently when they arrive at the edge of the paper.
- You should think about how you are going to represent directions in the turtle. It may simplify the design to represent *north* .. *northwest* as the integers *0* .. *7*.
- When you write the reflecting turtle you should consider how it behaves in all cases - for example at the corners and at the other non-corner locations when it approaches at 90 degrees or at 45 degrees.

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

- Make sure your main program class file **TurtleInterpreter.java** is at top level (i.e. not in a package subdirectory) while your six support classes **Turtle.java**, **BouncyTurtle.java**, **ContinuousTurtle.java**, **WrappingTurtle.java**, **ReflectingTurtle.java**, and **Paper.java** are in *package turtleSupport*, as described earlier.
- Submit these six Java files in the usual way using the **CATE** system.

Assessment

Command line interpreter	1
Turtle movements	3
Displaying content of paper	1
Design, style, readability (credit will be given for simple code and well designed data structures)	5
Total	10