COMP16212 Laboratories
Introduction

John Latham

January 2019

COMP16212 has 16 laboratory sessions, forming 10 exercises each of 1, 2 or 3 sessions duration. The marks associated with each exercise are weighted equally per session.

Thus a 1 session exercise counts as $1/16$ (6.25%) of the whole COMP16212 laboratory mark, and a 2 session exercise counts twice as much.

However, we must reserve the right to adjust this weighting for good reason if, for example, a particular exercise goes badly. We may even adjust the number of sessions for good reason.

In addition, there are three exercises – ex11o, ex12o and ex13o – which are optional. This means the following.

- If you do not do an optional exercise, it does not count.
- If you do an optional exercise, but get a lower average than your other marks, it is not counted.
- If the marks obtained for an optional exercise would increase your overall average, then they are counted. Each optional exercise counts as one more session, and so if all are counted, then each single session exercise is worth $1/19$ of the whole COMP16212 laboratory mark.
- The optional exercises are done in parallel with the non-optional ones, or after if you work ahead: they all have the same deadline as ex10.

1 List of exercises

You will find some of the exercises different in nature to those of COMP16121, in that they consist of one large task instead of lots of small ones. This will give you experience of developing larger pieces of software.

The following exercises form the COMP16212 laboratories.

Number	Duration	Description	Page
1.3D	3 sessions	The Snake Game Java Case Study	5
2.2D	2 sessions	Inheritance: Lottery Extra	21
3D	1 session	Selected book coursework tasks	-
4D	1 session	Selected book coursework tasks	-
5.2D	2 sessions	Selected book coursework tasks	-
6D	1 session	Selected book coursework tasks	-
7.2D	2 sessions	Selected book coursework tasks	-
8D	1 session	Selected book coursework tasks	-
9.2D	2 sessions	Selected book coursework tasks	-
10D	1 session	Selected book coursework tasks	-
11oD	0 sessions	Ordered Binary Trees	41
12oD	0 sessions	Flight booking	51 (Web version only)
13oD	0 sessions	MP3 player	63 (Web version only)

See on-line for the specific tasks associated with exercises 3D to 10D. Please note that, unlike COMP16121, there will be tasks in the book which do not appear in these exercises. At the time of printing this manual, the list of tasks per exercise is as follows.

This may well be changed – check on line.

Exercise	Tasks
3	15.6, 15.8
4	18.2, 18.3
5	18.6, 18.7
6	19.3, 19.4, 19.5
7	20.4, 20.5, 21.2, 21.3
8	21.4, 21.5
9	21.8, 22.5
10	22.8

Note the exercises that use more than one session may have intermediate deadlines. These will simply be a tick to say your work has reached a certain stage by that time, and each one will be worth 10% of the corresponding exercise. ***You must* not *run* `submit` *for these intermediate deadlines.***

2 Organising your work

You are expected to continue to organise your work as you did for COMP16121. For the task oriented exercises, this will be precisely the same – including the use of test scripts. (Refer to the laboratory manual for COMP16121 in the unlikely event that you have forgotten these details.)

It is worth reminding you of our right to remove your submissions from the archive if you do not get work marked as soon as you can after finishing it. *Please do not delay – in laboratory times, marking of completed exercise must take priority over working on new ones.*

3 Logbook

You are expected to continue using your logbook throughout the COMP16212 laboratory, and its use will form part of the assessment. By now you should have a fairly good idea of what logbook entries are useful to you, and appreciate more that the details depend on you as an individual, with your relative experience and the way your mind works.

In particular, you probably ought not to be writing full code in your logbook – you never were asked to even in COMP16121.

Now that the tasks are harder you may well find it useful to list any requirements that you do not understand on first reading of the exercise specifications, and then later, after you have clarified the issues, write your own explanations of them.

4 Working alone

Programming is an instance of design, and as such is a mostly creative activity. You can only learn creative skills by practising them. You can only be *taught* concepts about the raw material building blocks (e.g. a programming language), and perhaps some tips about good approach to organising your thoughts. The rest you must *learn* yourself.

The COMP16212 exercises are designed to be undertaken individually. It is most important that you work on your own, obtaining only very basic help from your fellow students. If you work together, then you will seriously undermine your own learning. As you know, such behaviour is regarded by us as academic malpractice, and you and your co-workers will lose all marks when we catch you.

If you see someone else's code, or detailed design, before you have completed your own, then you have cheated. Worse still, they have put your learning in jeopardy by showing it to you: that is not really the act of a friend.

If you do want to work together on some exercises, then please do, but not with the laboratory exercises! Instead pick any of the exercises from the books (and which we are not using in the laboratories) to work together on. This can be a beneficial way of learning, depending on who you are working with.

However if you treat the laboratory exercises in this way, any benefits of working together are far outweighed by the disadvantages to you. There is a good chance that you cannot see this fact yet, particularly if you are used to an environment where you have always worked with others. You should accept that the *kind* of learning you are now being asked to achieve is different in many ways to what you may be used to, and so the *way* you learn must also be different.

4.1 The bottom line

Our rules on collaboration/plagiarism are there to deter you from trying to learn in the wrong way, in case you don't yet see why you shouldn't work together, and thus protect you by brute

force if necessary! Be warned: we mean it. Unfortunately, the collaboration/plagiarism detection process is an ongoing one, and it is not always possible to detect academic malpractice immediately after work is handed in. Sometimes it is only detected much later, perhaps near to the end of the year, when it is too late for those caught to do extra work in order to pass. Sadly, there are sometimes a few students who do not make it to the second year, apparently because they thought they could get away with cheating, and ended up at the last minute with failed laboratories. I personally wish I could be better at convincing people how serious this issue is.

Worried? Please don't be. If you never ever see (or hear...) the code, or detailed pseudo code or design of another student, before you complete your own work; nor help another student at such level of detail, then you have nothing to worry about.

5 Meet the deadlines: don't be obsessed about getting full marks

We intend that none of you should need to miss any of the deadlines, unless you have some special circumstances like illness. It is true that you may need to work hard on the exercises to meet the deadlines, and you should be prepared to do this: *no pain, no gain!*

However, due to the extremely wide range of current abilities and previous experiences, we do not expect every student to do every part of every exercise. Instead, the exercises typically have a number of places at which you can stop working. So, you should work hard, get as far as you can, and stop when you run out of time or when you complete the whole exercise. That way, your learning will be timely – to tie in with the lectures, and at a depth that reflects your growing ability. Moreover, you will not get depressed about falling behind.

On the other hand, you might get depressed about not getting the highest marks available, if you let yourself. You must accept this fact: half of our students have suddenly moved from being "near the top of the class" to being "below average", simply by coming to University. You should bear in mind that a good student is not somebody who gets high marks, but somebody who works hard and knows at any time what their ability is. That way, the good student can improve.

Even if you don't buy this argument yet, there is a simpler one which you will. Given that the penalty for missing a single deadline is probably in effect to lose all the marks you get for it, you are *nuts* if you decide to miss a deadline in order to get more marks for it! ;-)

COMP16212 – Exercise 1 (3 Sessions)
The Snake Game Java Case Study

John Latham

1.1 Aims

The main aim of this exercise is to give you practice with programming and algorithm development, especially in the manipulation of two dimensional arrays.

You should also gain a deeper insight into the benefit of *design* and incremental development. Much of the design has been done for you, and this helps you develop a relatively large piece of code in stages.

In addition, you will be exposed to the idea of using program code to test the behaviour of program code. This is achieved by us giving you a test suite, written in Java, which uses a testing framework called JUnit. This test code was written by Suzanne Embury, who teaches on the second year software engineering course.

1.2 Learning outcomes

On successful completion of this exercise, a student will:-

- Have an increased experience of working in a context where there is a lot of code already written.
- Have an increased experience of working to a given set of requirements and to a specified interface.
- Have written code handling a two-dimensional array.
- Have an increased experience of writing code, especially using loops and if statements.
- Have been exposed to the idea of unit testing via the use of test code.

1.3 Introduction

This laboratory is associated with the snake game case study, and involves you attempting to complete the game. As you recall, the rest of the program has been developed already. You must now develop the class `Game`, this being the core model class of the program.

You will need your copy of the case study handout for reference.

1.4 Stages of development

The development work is divided into fifteen parts and sub-parts, including one containing some suggested advanced extras. There is also a part to look more closely at, and fix, the unit test code. It is *not* expected that everyone will complete all the parts: that will only be possible if the parts are too easy (I don't think so!) or if most people cheat (at their peril...!).

We expect that all of you will get to the end of Part D, and about half of you will get to the end of Part E. The parts have been identified to help your development, but also to make it possible to mark your work within the limited marking resources available. Please undertake your development of the parts in the order listed, or you may end up creating extra work for yourself and/or result in a piece of work which cannot be marked.

Each part will be marked out of its maximum, on a qualitative basis with the usual emphasis on choice of identifiers, layout, and comments, as well as suitability and correctness of the code.

So the idea is for *you* to get as far as *you* can, in the time available, *without needing to obtain unhealthy help from others*. It is a learning exercise, not a mark collecting exercise. Remember: if you see someone else's code, or pseudo code, before you write your own version, then you will be unable to write it any other way. You (and they) have *cheated* and you (and they) will be given zero when you are caught. It is extremely bad for your learning to obtain this kind of so-called help, as you are almost certain to obtain a false sense of your ability: either dangerously high (so you will fail later) or debilitatingly low (so you cannot progress now). It is bad enough to jeopardise your own learning in this way, it is even worse to offer such so-called help to a so-called friend.

By this definition, if two or more of you work together then you have cheated and you will all be given zero.

So, get only healthy help, work hard, work alone, get as far as you can, *and meet the (extended) deadlines.*

Part	Description	Marks
A	The score message	5
B	Constructor and grid accessors	10
C1	`setInitialGameState()` method	10
C2	Place food	5
C3	Place snake	10
D	Set snake direction	5
	Subtotal	45
E1	Move method	5
E2	Move the snake head	5
E3	Move the snake tail	5
E4	Check for and deal with crashes	5
E5	Eat the food	5
	Subtotal	70
F	Cheat	5
G	Trees	10
H	Crash countdown	5

Part	Description	Marks
	Subtotal	90
H+	Fix the JUnit tests	10
I	Advanced extras	10
	Total	110

In addition, the *appropriate* use of your logbook is marked out of 15, making a total of 125 marks. Please do not fake entries after the event – design and planning for each part is done in advance of implementing each part, and testing and reflection is done after it is implemented.

Use of your logbook includes you recording the fact that you have run the unit tests immediately after each stage (as explained below), along with any observations or bugs found, etc.. If this is missing, or looks like it was not written at the time *you will receive zero for the logbook marks*.

The stages are explained in more detail shortly.

1.5 Two deadlines

There are two weeks allocated to this exercise, a total of three laboratory sessions. There are deadlines at the end of the first and last sessions. *The first deadline cannot be extended* (apart from for special circumstances). The other deadline is extendible to the *strict start* of the following session in the usual way. The deadlines are as follows.

1. Convince a demonstrator that you have (essentially) completed at least Parts A and B listed in the table above. *This deadline cannot be extended.* The process of convincing a demonstrator will not take long – you will simply run your program and he or she will check it and look at your code. ***Do not produce a printer listing for this stage, nor run `submit`.*** The stage is a tick deadline (marked out of 1), but it counts 10% of the exercise.
2. Get as far as you can with the rest of the stages, stopping when your time runs out (or before then if you wish). You will print off your work and have it marked in the usual way.

1.6 The laboratory exercise

You should work in your `$HOME/COMP16212/ex1` directory.

Copy the file `/opt/info/courses/COMP16212/ex1/Game.java` to your directory. (Please don't use a file browser!) This contains a skeleton `Game` class already. The file `/opt/info/courses/COMP16212/ex1/Snake.jar` contains the rest of the program, which has already been compiled. In order to compile your part of the program, you will need to provide a `classpath` argument to `javac` which tells the compiler where to look for ready compiled classes. Write yourself a shell script called `compile`, containing the following text.

```
#!/bin/bash
javac -classpath /opt/info/courses/COMP16212/ex1/Snake.jar:. Game.java
```

The `classpath` argument tells `javac` to look for compiled classes first in the archive file `/opt/info/courses/COMP16212/ex1/Snake.jar`, and then in the current directory. The colon (`:`) is a separator between items in this list of paths.

Write another script, called `run`, to run the program. You will want this to be able to take command line arguments (to set the size of the grid), so you should make it contain the following text.

```
#!/bin/bash
java -classpath /opt/info/courses/COMP16212/ex1/Snake.jar:. Snake "$@"
```

Any arguments passed to the script will be passed on to `java` – `"$@"` means 'all command line arguments, with quotes round them in case they contain spaces'.

Compile and then run the program.

```
./compile && ./run
```

You should see a window containing a black box, with a white bar containing a simple message. The black box is the area where the field will be seen, and the white box is the score message bar. Try pressing 'h' (after you have moved the mouse into the window). You can see all the interface is there ready to go, you can even change the speed of the speed controller. Press 'r'. The game is running, it is just that the `move()` method of the `Game` class is doing nothing.

Look at the source for the class `Game` (e.g. use the `less` command). You will see it already contains the method headers and is divided into the parts for you. *Please do not reorder these parts in your answer.* Whilst, arguably, a different final ordering might or might not be better from a professional programming practice point of view, the order given here is critical for the process of marking: it must be easy for your marker to see how far you got. So, if you changed the order of your work, you would be creating a significant amount of extra work for us – except that we might just refuse to mark it instead! :))

1.6.1 Method names: Warning

When editing the file, you should take care not to alter the names, or parameters, of the public methods. If you do, you will not get any warning from the compiler but you will get a run-time error when the pre-compiled part of the program tries to call the missing method.

If this happens to you, you can check your file for missing methods by running `/opt/teaching/bin/SnakeGameMethodCheck`.

1.6.2 Part A: the score message

In this first part, you will need to create a variable for storing the score message. Should this be private or public? Now alter the accessor and mutator methods to use this variable appropriately. Finally, make the `getAuthor()` method return your name.

Now compile and run the program. You should see a more meaningful message in the score message bar, including your own name. You should also see your name in the title bar of the

window.

Next you must test your implementation using the test suite we have provided (written by Suzanne Embury). To keep this simple to use, we have provided a command (which lives in `/opt/teaching/bin`). You will want to run this after you have completed each part so you can test all the parts completed so far. You specify the part you have just completed as a command line argument.

```
testMySnake A
```

After a short while with messages whizzing past, this should start firefox showing the results of the test. In the main area it will show you how many tests were performed, how many failed and how many resulted in an error.

In JUnit, a failed test is one that was expecting a certain behaviour, but found a different one.

An error is when something has gone wrong which was not even expected. E.g., if you were to attempt to test parts that you have not yet implemented then you probably would get some errors.

Click on the number under the heading 'Tests' and you will see the results of the two tests which have been performed for part A.

One, called `shouldReportTheScoreMessageWeSetForTheGame` will probably have status 'success'. This means the behaviour it expected was found, and suggests that your score message setting and getting has worked properly. (If it has failed, you need to fix your code!)

The other, `shouldReturnCorrectAuthorsName` has failed. A test may fail because the code being tested is wrong, or because the test is wrong! It is the latter which applies in this case. You will find many tests in the test suite which fail like this. These all have the form

```
Expected: is "<INSERT some text HERE>" but: was "some other text"
```

This is because the test suite does not know what text strings you have placed in your code (in this case, your name) and the test code will need to be edited to fix that. *Leave this for now* – in a later part of the overall exercise you may address them (if you get that far).

Record the fact you have run the tests in your logbook, along with any observations notes or details of bugs that were found, etc..

Quit the firefox window when you have finished looking at the results.

1.6.3 Part B: constructor and grid accessors

In this part you will create the grid of cells and its associated accessor methods.

You will need two variables, one to store the grid size, and the other to store the grid. The latter will be a two-dimensional array of cells. Both of these variables should be `final`: once their values are set they should not be able to change.

The constructor of the class is given the required grid size, which you will want to store. You will want to create the array and assign it to your grid variable, and then fill it up with new cells, using two nested loops.

The two accessor methods `getGridSize()` and `getGridCell()` need to be altered to return the correct results.

Now compile and run the program. You should see the box is full of cells shown as the word `"NEW!"` written in yellow on a green background. The `CellImage` class shows a newly created cell in that way. There should be eighteen rows by eighteen columns. Check your implementation is correct by running your program with a command line argument to choose a different grid size, as follows.

```
./run 5
```

Now apply the unit tests for this part.

```
testMySnake B
```

You should now be able to see the tests for parts A as before, plus a further 3 tests for part B. Hopefully, the new tests have a 'success' status – if not, fix your code!

Don't forget to update your logbook.

1.6.4 Part C: initial game state

Now for something a little more challenging. In this stage you are going to develop code to initialise the game, including placing the snake and the food. You should split this part into the following sub-parts.

1.6.5 Part C1: `setInitialGameState()` method

In the first sub-part, you will develop the code for the `setInitialGameState()` method. This will invoke two private methods which you will write in the next two sub-parts, one to place the food and the other to place the snake.

The `setInitialGameState()` method must first make all the cells in the grid get the clear cell type, using two nested loops and the `setClear()` method. Note: you do not *ever* want to create new cells now, only change the state of the ones you have already made. This is because the class `GameImage` will have already linked the cell images to the cells made when the game was constructed.

Now try running the program. The box should just show clear cells with a green background. Hopefully there are still eighteen rows and eighteen columns, but you cannot see them.

The final part of the `setInitialGameState()` method involves calling the methods to place the snake and the food, and setting the score to zero. (Note – your code should call the method to place the snake before the one to place the food, otherwise the snake could overwrite the food!)

To compile the code for `setInitialGameState()`, you will need a variable to keep track of the score and **stubbed** versions of the two private methods. The method to place the snake will need to be given all the arguments which were passed to `setInitialGameState()`.

Now apply the unit tests for this sub-part, fix your code if required, and update your logbook.

`testMySnake C1`

From now on, this document will not remind you every time to apply the unit tests for each (sub) part, but you must do so, and also update your logbook as you proceed.

1.6.6 Part C2: place food

Next you should develop the method to place the food in the field. This will need to find a randomly chosen clear cell, and then set the type of that using the `setFood()` method of the class `Cell`.

For simplicity, you can assume there is at least one clear cell in the field. So, perhaps the easiest approach is to use a **do-while** loop: inside the body choose a cell at random, and continue the loop if the type of that cell, obtained via `getType()`, is not `Cell.CLEAR`. To choose a cell, you will need to choose its X and Y values randomly. These must lie in the range zero to one less than the size of the grid.

To obtain two integers in the range 0.0 to $n-1$, you can use the following code.

```
x =  (int) (Math.random() * n);
y =  (int) (Math.random() * n);
```

You should use appropriate names for `x`, `y` and `n`.

Here is some pseudo code to help you.

```
do
  get a random value for the food X position
  get a random value for the food Y position
while the cell at the chosen position is not clear
set the cell at the chosen position to food
```

The variables used to store the X and Y positions of the food can either be declared within the method, or be private instance variables of the class. If you declare them outside the method, then you will be able to know where the food is if and when you come to the advanced extra feature of making the food run away from the snake. You could declare them within the method for now, and move them outside it later, if you wish. If you do declare them outside the method, please put their declaration just before your method to place the food, so as to make marking easier. (That is arguably a good place for them to be declared anyway.)

Compile and run the program, to check that the food appears. Press the ‘a’ key to start the game again, each time you do the food should disappear and reappear at a random position. The ‘a’ key is causing your method `setInitialGameState()` to be called, so if it does not work you know what to do.

Later on, if you have time, you can come back to this code and improve it so it does not assume there is a clear cell. As you can see, if there was no clear cell then the loop would continue searching forever. This is only likely to occur in a game using a very small grid size.

(Don’t forget to also run the unit tests.)

1.6.7 Part C3: place snake

The final sub-part of game initialisation, is the method to place the snake. If you have stubbed this right, it should be given four integer arguments. The first two specify the X and Y positions of the tail of the snake. The third argument states the length of the snake. The fourth one indicates the direction it is facing.

You may assume that the snake will fit in the field: the code which calls your `setInitialGameState()` method has already ensured these values are sensible.

You will need to add six instance variables to your class to record the following information: the direction of the snake, the X and Y positions of the tail of the snake, the X and Y positions of the head of the snake, and the length of the snake. As always, take care to choose appropriate names for these variables. To make marking easier, please put these variable declarations just before your method to place the snake.

Placing the snake has mainly three parts to it: placing the tail, placing the body and placing the head. Additionally, you will want to store the snake direction, its length and the positions of its head and tail.

To place the tail you must set the type of that cell to be a snake tail. The snake out direction in the cell will be the direction of the snake, and the snake in direction will be the opposite direction. Don't forget about the `opposite()` method of the `Direction` class.

Run the program to check you have the tail of a snake showing. The laboratory version of the game chooses at random one of eight possible start states. The snake should appear in one of the four corners of the field, facing in one of the two directions away from the corner. Press the 'a' key several times to start the game again – each time one of the eight possible start states will be chosen.

To place the body of the snake will require a loop. This must execute `requiredLength - 2` times, placing one cell of the body each time. You will need two method variables to store the X and Y values of the next cell to be turned into a snake body. These will start off addressing the cell which is next to the tail, and move along one place each time. You will find the methods `xDelta()` and `yDelta()` from the `Direction` class very useful for this.

The following pseudo code should help you.

```
next cell position = the cell after the tail

loop requiredLength - 2 times
  make the cell at next cell position be a snake body
    with directions: snake in = opposite of snake direction
                     snake out = snake direction
  next cell position = the cell after the next cell position
end loop
```

Run the program to check you have the body appearing now. You should get seven cells of body after the tail in the given direction. Press the 'a' key several times to make sure it works with all eight possible start states.

To place the head of the snake, you must simply set the type and snake directions of the cell in

the next cell position. The variables you used for the next cell position in the loop should already have the right values.

Make sure you have stored the required values in the six instance variables you declared at the start of this sub-part.

Run the program to check you have the whole snake appearing now.

Finally, run the program again with a grid size argument of 10 to check that you get a snake with a length of five.

(Don't forget something else too – last reminder!)

1.6.8 Part D: set snake direction

This next part is a little simpler than the previous. The method `setSnakeDirection()` is called from the game interface with the new required direction given as an argument. You may assume this is one of the four legal directions, as the game interface will only ever call this method with a legal value. You can also assume that the method will only be called after `setInitialGameState()` has been invoked at least once, i.e. that there is a snake somewhere in the field.

Your method will want to set the snake out direction of the head of the snake, to the direction given in the argument. (It will not want to change the snake in direction of that cell.) This is easy, as you know where the snake head is at any time, because you created two instance variables to keep track of its X and Y values. There is one complication though. A snake is not allowed to make its head face straight back along its body. This is most likely to be 'requested' by the player when he or she is trying to undertake a U-turn, but has not allowed enough time for the snake to move after the first of the two changes of direction. So, if the required direction is the same as the snake in direction of the snake head, you should not permit the direction change, but instead put a suitable message in the score message variable.

Run the program to check you can move the head of the snake. Check that your score message appears when you try to make the snake face backward. A good way to clear the message again is to press 'r'.

When you run the unit tests for this part, you will get another 4 failed tests which are not yet set up to expect the correct message.

(When you do the thing I'm not reminding you to do, you'll get lots more failures like the one in part A. Again, don't worry about fixing that for now, as long as you understand what is causing them.)

1.6.9 Part E: snake movement

This part involves making the snake take its move. You should develop it as the following sub-parts: the move method itself, moving the head, moving the tail, dealing with crashes and eating the food.

1.6.10 Part E1: move method

In this first sub-part, you will develop the code for the `move()` method. This will invoke four private methods which you will write in the next four sub-parts.

The snake should only move if the head of it is not bloody, so you should check, and do nothing if it is.

The first step in making the snake move is to obtain the new position of the head. That position should then be checked for crashes – is it off the grid, or is it occupied by another part of the snake? If the new position is okay, then the move can go ahead. First the head of the snake moves to the new position. Then, either the tail moves along, or the snake gets longer because it has just eaten the food. So, before moving the head into the new position, it is important to remember whether there was previously food at that cell.

The following pseudo code should help you.

```
if snake head is not bloody
  compute new position of the head

  it is okay to move = check for a crash at the new position and deal with it

  if it is okay to move
    remember whether there is food at the new head position

    move the head to the new position
    if there was food at the new head position
      eat the food
    else
      move the tail
    end if
  end if
end if
```

You should write stubs again for the four sub-part methods. The method to move the snake's head could be given the position of the new head, rather than compute it again. The method to check for and deal with any crashes could also be given the new position. It must return a boolean, `true` if there was no crash and the move can go ahead, `false` otherwise. The stubbed version should just return `true`. The method to eat the food must be given the move value, as passed to the `move()` method, so it can update the score appropriately.

Compile the program to check for syntax errors etc. You can run the program, but it won't do very much more than the previous part yet. Note what happens if you point the head into the side of the field and start the snake's movement by pressing 'r'. Check in the xterm from which you ran the program. Can you explain this?

1.6.11 Part E2: move the snake head

Now develop the body of the method to move the head of the snake. This needs to make the existing head cell into a snake body, and the new position into a snake head. What should the snake in and snake out directions be for these two cells?

Run the program to make sure it works properly. If you crash off the side of the grid then your program will fail, as you haven't done any crash detection yet. The snake should appear to grow on every move. Did you remember to update the variables storing the position of the snake head?

Make sure the snake looks right when it turns round corners.

1.6.12 Part E3: move the snake tail

Your next job is to make the tail move too. Essentially, this involves clearing the current snake tail cell, and turning the next cell along the snake into a tail cell. You can find the next cell along the snake by following the snake out direction of the tail.

1.6.13 Part E4: check for and deal with crashes

There's two possible ways the snake can crash: either it can go off the side of the grid, or it can crash into itself. This method could be given the proposed new position of the snake head, check it is not off the side, nor already occupied by a piece of the snake. If there is such a problem, then the method must return false as its result, but also it should make the snake bloody and put a message into the score message indicating the problem.

If the snake has attempted to leave the grid, then its head should go bloody and the message should be something which indicates that the snake cannot leave the field.

If the snake has crashed into itself, then its head and the part it has crashed into should both go bloody, and the message should be something which indicates that the snake cannot eat itself.

Run the program to make sure it works, by crashing off the side of the grid and into the snake. No error messages should be printed to the xterm in either case.

1.6.14 Part E5: eat the food

In this part you will develop the method to eat the food. You should also make the method `getScore()` return the correct score.

When eating the food, the length of the snake increases by one, the score should also be increased, there should be a message to the player, and more food should be placed in the field.

The score should be increased by this formula:

```
moveValue * ((snakeLength / (gridSize * gridSize / 36 + 1)) + 1);
```

where `moveValue` is the argument passed to the `move()` method by the game interface.

The message to the player should be something which indicates how much the score has just been increased.

1.6.15 Part F: cheat

In this part you develop the code for the `cheat()` method. Every time this method is invoked the score should be halved, and the user should be informed that he or she has lost half the score, and how much that is. Also, all bloody cells in the field should be made not bloody. You can do this with two nested loops searching for all bloody cells.

1.6.16 Part G: trees

This is the first part in which you will probably have to go back and alter some of your previous parts. It's a very good idea to keep a copy of your work up to now, just in case you decide to stop working.

The trees feature makes the game much more of a challenge for the player, and is not terribly complicated to implement. However, it does affect a number of existing places in the program.

You will need an instance variable to store the current number of trees in the field, and another to record whether trees are currently enabled or not. Please declare these variables in the area marked as Part G.

The `toggleTrees()` method essentially switches trees on if they are off, or off if they are on. When switching them off, you will need to remove all the trees from the field. You can do this with two nested loops. When switching them on, you will want to plant a single tree in a clear cell. You should do this through a separate private method. This method can work in a similar way to that used to place food. You may assume, for simplicity, that there is at least one clear cell.

You will need to make alterations to your existing code. In the method `setInitialGameState()` you will want to set your number of trees to zero. Also, if trees are currently enabled, you will want to plant a single tree, after clearing all the cells.

Also, in your method to check for and deal with crashes, you will need to add code that checks whether there is a tree at the proposed new head position. The snake must die if it hits a tree. You will want to give an appropriate message to the player for this case.

Finally, in your method to eat the food, you must add code to deal with the case that trees are enabled. If they are then the score increase gained for eating the food must be multiplied by the number of trees in the field. The message to the player should indicate the raw score increase, the number of trees, and the actual score increase. Another tree should also be added to the field.

1.6.17 Part H: crash countdown

This part involves you writing code that causes a delay before a crash of the snake will actually kill it. You will need two new instance variables. Please declare them both in the area marked for Part H. The first should be a final integer assigned with the number of moves which is the crash countdown start value. The number 5 is recommended. The second variable should be an integer showing the current countdown value.

You should write a private method to reset the countdown to the countdown start. If, prior to this reset, the current countdown is less than the countdown start, then the method should indicate to the player that they have been lucky to escape death, and by how many moves. This method will be called every time a move has been successfully made by the snake.

You should also make another private method which reduces the current countdown by one. If after this decrement it is still greater than zero, the method should indicate to the player they have only so many moves before death and that they should move away quickly. The method should return the result `false` to indicate that the snake is not yet dead. On the other hand, if the countdown after decrementing has reached zero, then the method should reset it to the countdown start, and return `true`, without setting a message for the player. This method will be called every time a move would cause a crash.

Now you will need to alter your existing code to plug these two new methods in. Your method to check for and deal with crashes will need to call your method to count down the countdown variable if the move cannot go ahead. It should still return `false` if the move cannot go ahead, but only set the snake bloody if the countdown method indicates the countdown has reached zero. The specific messages indicating the reason for the death of the snake should only be shown when the snake actually dies – the countdown method will cause a warning to be shown otherwise. You may like to restructure your original method to check for and deal with crashes, introducing method variables to help separate crash detection from the killing of the snake. One such variable might store the string you will put into the score message, if (and only if) the countdown has run out.

Your other new method for resetting the crash countdown, will be called from your method to check for and deal with crashes, but only if the move *can* go ahead.

This countdown resetting method should also be called by the `cheat()` method.

1.6.18 Part H+ – Fix the JUnit tests!

Now that you have completed the non-extra development, you can focus on the unit testing code and in particular fix all the faulty failures.

When you first ran `testMySnake`, it will have created a sub-directory called `UNIT-TESTING` within your exercise directory.

```
$ ls UNIT-TESTING/*.java
UNIT-TESTING/Game.java              UNIT-TESTING/GameTestsForExE2.java
UNIT-TESTING/GameTestsForExA.java   UNIT-TESTING/GameTestsForExE3.java
UNIT-TESTING/GameTestsForExB.java   UNIT-TESTING/GameTestsForExE4.java
UNIT-TESTING/GameTestsForExC1.java  UNIT-TESTING/GameTestsForExE5.java
UNIT-TESTING/GameTestsForExC2.java  UNIT-TESTING/GameTestsForExF.java
UNIT-TESTING/GameTestsForExC3.java  UNIT-TESTING/GameTestsForExG.java
UNIT-TESTING/GameTestsForExD.java   UNIT-TESTING/GameTestsForExH.java
UNIT-TESTING/GameTestsForExE1.java
```

The first file listed above is a copy of your code, made when you last ran the test script. The remaining ones contain the source code for the unit tests, and as you can see they are divided into a separate class for each (sub) part.

Your job is to work through those as needed to alter them to that faulty tests no longer fail. You will see that the code contains some Java concepts that we have not explained in the course. You wil lalso see that it uses some external packages which you have not met before. *It is important that you do not feel the absolute need to understand every line of code.* Despite not understanding everything, you should be able to find the places where you need to edit the text strings to match the ones you choce in your code.

Along the way, you are being exposed to the general idea and principle of unit testing via dedicated code, and that is sufficient for now for your learning. However, you can if you wish try to find out more – but please do not feel that you have to at this stage.

1.6.19 Part I: advanced extras

And now for the advanced extras. In order to get full marks for this part, you will need to do well *all three* of the following suggestions. (Doing only one or two will receive proportional marks.)

- Gutter trails: a feature toggled on and off by the ‘g’ key. When on, the snake leaves a trail in the grass which slowly fades. There is a type of cell you can use for this. Levels of 50 down to 0 are recommended. You will need to check for such cells and decrement their level at the start of each move. You should make a private method in the area marked Part I, and call it from the `move()` method. You should set such a cell when the tail moves, if the feature is switched on.

- Burn trees: a feature invoked by pressing ‘b’. This will cause a tree directly in front of the snake to immediately disappear. You will need to check that such a cell is not off the grid, and does contain a tree. Don’t forget to decrement the number of trees, otherwise scoring will be wrong.

- Food movement: a feature toggled on and off by the ‘m’ key. When on, the food will run away from the snake head. You should write a private method in Part I to move the food, which will be called at the end of the `move()` method. It is surprisingly simple to make the food move in an apparently intelligent way, and one algorithm is as follows. The food has a direction which it is moving in. On each move the food either moves in that direction, or it changes direction by turning right (or left if you prefer, but it is best to be consistent

each move). It will move if the cell it would move to exists, is clear and is not geometrically nearer to the snake head than the food's current position. It turns otherwise.

1.6.20 Part J: optional advanced extras

Bored? Cannot sleep at night? Find it all too easy? Well, here are some more suggested advanced extras that are *not* in the marking scheme – you will not get marks for these, just self satisfaction and esteem. If you do try them, I recommend you do it in a way that still makes it easy to have your previous work marked: copy your files to a separate directory and work there for this continued development. Feel free to show off to your marker, but don't complain if he or she still wants to to give you less than full marks for the main stuff.

- Have a demonstration mode which can be toggled on and off, in which the snake makes its own moves. It tries to avoid crashing, but heads toward the food. If it does crash it causes the game to be initialised again.
- Have two snakes! The user can toggle in and out of two-snake mode by typing a key. Where does the second snake get placed? Perhaps going into two-snake mode causes the game to be initialised. When the game is initialised in two-snake mode, the second snake is placed in the opposite corner with the opposite direction. When in two-snake mode the user can switch between controlling them with another key. Maybe only the current snake moves, while the other one stays still. Or maybe, for a hard game, they both move on each move!
- Have more than one piece of food. Each time some food is eaten, two new pieces are created, and they all can move (if food movement is enabled).

1.7 Assessment criteria

The exercise is marked out of 125 marks, and is worth 18.75% of the assessment for the COMP16212 laboratory.

Each of the fifteen development parts and sub-parts will be marked out of their maximum, with a penalty for poor quality of code (including identifiers and comments). Then use of your logbook will be marked out of 15, proprotional to how many parts you implemented and appear to have tested at the time (using evidence in your logbook). And then your fixing of the unit tests. This gives a maximum mark of 125.

1.8 Completing your work

As soon as you have completed your work, you must run the program `submit` while you are in your `COMP16212/ex1` directory. *Do not wait until it is about to be marked, or ARCADE will probably think you completed late!* `submit` will only work if you have named your files correctly

– check for minor filename variations such as the wrong case of character. The required files are named as follows.

```
Game.java
compile
run
```

Then, as soon as you are next present in the School (e.g. straight away) run `labprint` and go immediately to the printer to collect your listing. When you get marked, the teaching assistant will write comments on the listing. *You must keep the listing in your portfolio folder afterwards.*

For the face-to-face feedback and marking process, you should do the following when your teaching assistant arrives.

1. Run `submit-diff` to show there are no differences in your work since you submitted it.
2. Have the feedback and marking process.
3. Run `submit-mark` to submit your mark. This will require the mark and a **marking token**, given to you by the teaching assistant.

1.8.1 Conclusion

Depending on how far you got, you might find your `Game` class seems large. Out of interest, my sample answer including the three advanced extras, has around 520 lines, excluding documentation. In world terms, this is not big – classes with more than 2000 lines are quite common in real programs. However, the `Game` class could be split into several classes, such as one to handle the grid, one to handle the snake, another to handles trees, etc..

If you wish, you can experiment with splitting up the class in this way. However, for ease of marking, *please do not split up the version you are handing in*. Instead, once you have finished the exercise, you could make a copy of your entire directory and experiment there.

COMP16212 – Exercise 2 (2 Sessions)
Lottery Extra

John Latham

2.1 Aims

The main aim of this exercise is to help you gain a deeper understanding of inheritance, particularly the overriding of methods.

2.2 Learning outcomes

On successful completion of this exercise, a student will:-

- Have written code which overrides a method in a superclass, and also calls the overridden method.
- Have written new subclasses of existing classes.
- Have gained some appreciation of component testing, by writing a separate test program for each part, rather than one big test program.

2.3 Introduction

This laboratory is associated with the lottery example in Chapter 16, and involves you making extensions to it. You will be working with a version of the program that has been progressed since Chapter 16, in that much of the GUI has now been developed, and the model classes have been altered to link in with the GUI. Some of the model classes will be further altered by you. You will also be writing some new classes.

You will need your copy of Chapter 16 for reference.

2.4 Stages of development

The work is divided into 5 parts. It is *not* expected that everyone will complete all the parts.

We expect that all students will complete Part A, nearly all complete Part B, about half complete Part C and about a quarter complete Part D. Only a few are expected to try Part E.

The parts have been identified to help your development, but also to make it possible to mark your work within the limited marking resources available. Please undertake your development of the parts in the order listed, or you may end up creating extra work for yourself and/or result in a piece of work which cannot be marked!

Each part will be marked out of its maximum, on a qualitative basis with the usual emphasis on choice of identifiers, layout, comments, suitability and correctness of the code and *appropriate* use of your logbook. (Faked logbook entries will not be marked, and will be reported as cheating.)

So the idea is for *you* to get as far as *you* can, in the time available, *without needing to obtain unhealthy help from others*. It is a learning exercise, not a mark collecting exercise. Remember: if you see someone else's code, or pseudo code, before you write your own version, then you will be unable to write it any other way. You (and they) have *cheated* and you (and they) will be given zero when you are caught. It is extremely bad for your learning to obtain this kind of so-called help, as you are almost certain to obtain a false sense of your ability: either dangerously high (so you will fail later) or debilitatingly low (so you cannot progress now). It is bad enough to jeopardise your own learning in this way, it is even worse to offer such so-called help to a so-called friend!

By this definition, if two or more of you work together then you have cheated and you will all be given zero.

So, get only healthy help, work hard, work alone, get as far as you can, *and meet the (extended) deadline!*

Part	Description	Marks
A	`getClassHierarchy()` method	10
B	`DramaticMachine` and `DramaticGame` classes	15
C	`MagicBall` class	15
D	`MagicWorker` class	15
E	Extensions (mainly for fun)	5
	Total	60

The above marks include use of your logbook. These stages are explained in more detail shortly.

2.5 Differences from Chapter 16

In Chapter 16, we discussed in detail the first phase of the development, namely the model classes. We only briefly talked about the second phase, the GUI classes. In this laboratory, you will be working with a later version of the program in which some of the GUI classes have been completed. We can now obtain images of people, balls, machines and racks, and we have a general GUI framework for them, and a speed controller for the animated parts of the program (such as the images of a person flashing when he or she speaks).

The control interaction with the end user has not yet been developed, and so the program is tested by suitable code in a main method.

2.5.1 The `LotteryTest` class

Here is the `LotteryTest` program used for testing. This is not testing the individual features, which would be tested separately. It is just a sample test for the general framework, and to get an end-user oriented feel for the GUI.

```
public class LotteryTest
{
  public static void main(String args[])
  {
```

A `SpeedController` is given an initial speed when it is created, and a `LotteryGUI` is given a name and a `SpeedController`.

```
    SpeedController speedController
      = new SpeedController(SpeedController.HALF_SPEED);

    LotteryGUI gui = new LotteryGUI("TV Studio", speedController);
```

We create various `Person` objects, and add them in to the GUI. Their images become visible as we add them. As part of the testing of the `SpeedController` we here have a delay after each person is added. This is simply a pause in the execution of the thread. The bigger the delay argument, the longer will be the delay, on the other hand, the faster the speed controller, the shorter will be the delay.

```
    TVHost tvHost = new TVHost("Terry Bill Woah B'Gorne");
    gui.addPerson(tvHost);
    speedController.delay(40);

    AudienceMember audienceMember1 = new AudienceMember("Ivana Di Yowt");
    gui.addPerson(audienceMember1);
    speedController.delay(40);

    Punter punter1 = new Punter("Ian Arushfa Rishly Ving");
    gui.addPerson(punter1);
    speedController.delay(40);

    Psychic psychic = new Psychic("Miss T. Peg de Gowt");
    gui.addPerson(psychic);
    speedController.delay(40);

    AudienceMember audienceMember2 = new AudienceMember("Norma Lurges");
    gui.addPerson(audienceMember2);
    speedController.delay(40);

    Punter punter2 = new Punter("Al Winsom");
    gui.addPerson(punter2);
    speedController.delay(40);
```

```
Director director = new Director("Sir Lance Earl Otto");
gui.addPerson(director);
speedController.delay(40);

CleverPunter cleverPunter1 = new CleverPunter("Wendy Athinkile-Win");
gui.addPerson(cleverPunter1);
speedController.delay(40);

Worker worker = new TraineeWorker("Jim", 0);
gui.addPerson(worker);
speedController.delay(40);

CleverPunter cleverPunter2 = new CleverPunter("Luke Kovthe d'Ville");
gui.addPerson(cleverPunter2);
speedController.delay(40);
```

We create two `Games` and add them in to the GUI. Their images become visible as we add them. We also associate a `CleverPunter` with each game.

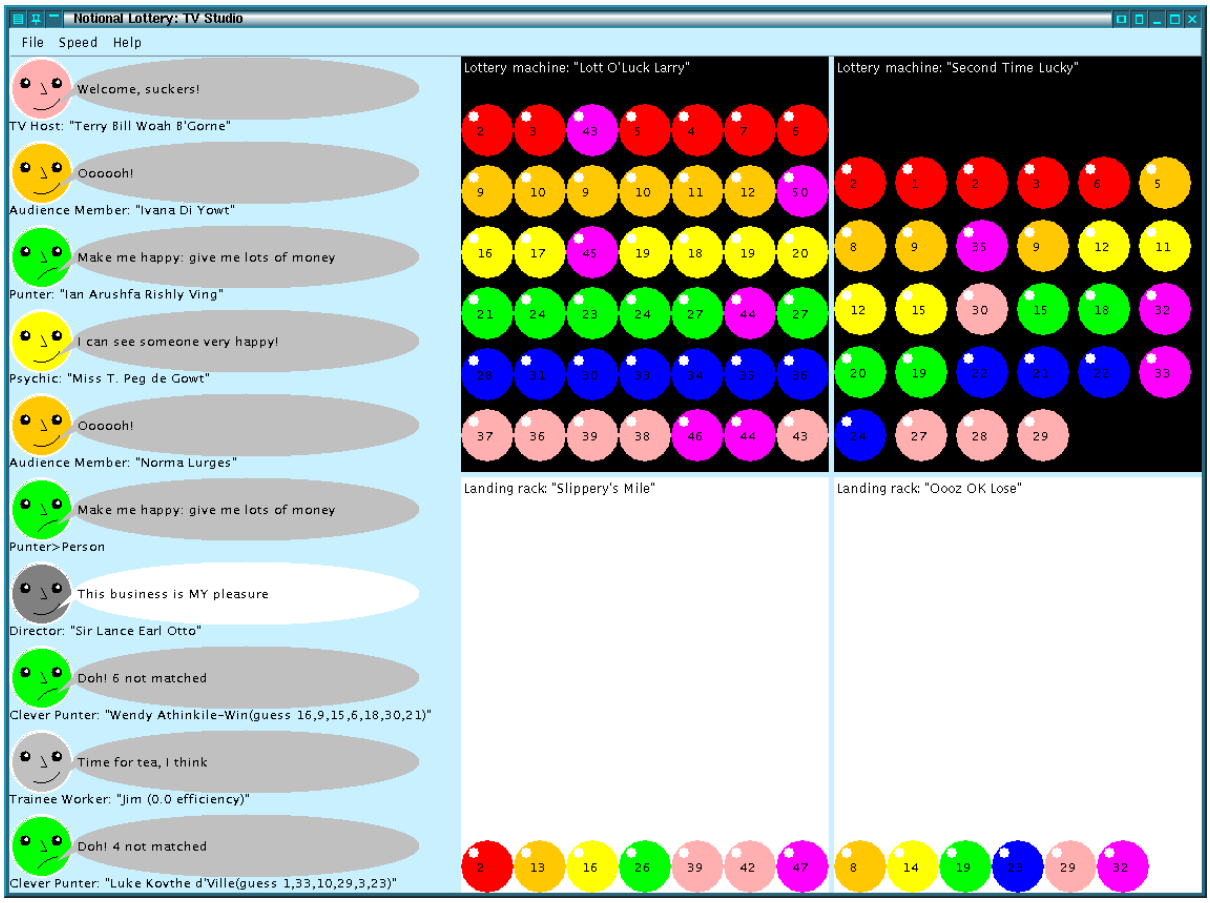
```
cleverPunter1.speak();
speedController.delay(40);
Game game1 = new Game("Lott O'Luck Larry", 49,
                      "Slippery's Mile", 7);
gui.addGame(game1);
speedController.delay(40);
cleverPunter1.setGame(game1);
cleverPunter1.speak();
speedController.delay(40);

cleverPunter2.speak();
speedController.delay(40);
Game game2 = new Game("Second Time Lucky", 34,
                      "Oooz OK Lose", 6);
gui.addGame(game2);
speedController.delay(40);
cleverPunter2.setGame(game2);
cleverPunter2.speak();
speedController.delay(40);
```

The rest should be obvious.

```
tvHost.speak();
speedController.delay(40);

worker.fillMachine(game1);
speedController.delay(40);
```

```
    punter1.speak();
    speedController.delay(40);

    for (int count = 1; count <= game1.getRackSize(); count ++)
    {
      game1.ejectBall();
      audienceMember1.speak();
      cleverPunter1.speak();
    } // for

    game1.rackSortBalls();
    speedController.delay(40);

    psychic.speak();
    punter2.speak();
    speedController.delay(40);

    worker.fillMachine(game2);
    speedController.delay(40);

    for (int count = 1; count <= game2.getRackSize(); count ++)
    {
      game2.ejectBall();
      audienceMember2.speak();
      cleverPunter2.speak();
    } // for

    game2.rackSortBalls();
    speedController.delay(40);

    while (true)
    {
      director.speak();
      cleverPunter1.speak();
      cleverPunter2.speak();
    } // for
  } // main

} // class LotteryTest
```

Here is a screen dump of the test program after it has reached the final endless loop.



2.5.2 The `Person` class and its subclasses

Here are the differences from the `Person` class in Chapter 16.

There is a new instance variable, to hold a reference to the `PersonImage` associated with this `person`.

```
private final PersonImage image;
```

The constructor creates a `PersonImage`, by calling a new instance method, `makeImage()`. The `PersonImage` object is given a reference to this `Person` object, which it keeps. Creating the image is done via a separate instance method, so that subclasses of `Person` can override it with one which creates a subclass of `PersonImage` if they wish to. (For example, `MagicWorker`, which you will create, needs to have a `MagicWorkerImage` instead of the more general `PersonImage`.

```
public Person(String requiredPersonName)
{
  personName = requiredPersonName;
  latestSaying = "I am " + personName;
  image = makeImage();
}  // Person
```

```
public PersonImage makeImage()
{
  return new PersonImage(this);
} // makeImage
```

There is an accessor method for the image.

```
public PersonImage getImage()
{
  return image;
} // getImage
```

There is a method which returns a representation of the class hierarchy of the person, from subclass to the `Person` class. This is a bit contrived, as it is not really a feature we would likely to want in the final game. It really only exists to support the first task in this laboratory. The `PersonImage` associated with the `Person` replaces the person's name with this text whenever the mouse is over the image. You will override it in the subclasses, so that for each kind of person it will return the class hierarchy for that kind of person. For example, the string returned by `getClassHierarchy()` in the `TraineeWorker` class should be `"TraineeWorker>Worker>MoodyPerson>Person"`.

```
public String getClassHierarchy()
{
  return "Person";
} // getClassHierarchy
```

There is an abstract method to return a colour which will be used by the `PersonImage` object as the colour of the person's face in the image. This is implemented in the subclasses `AudienceMember`, `CleverPunter`, `Director`, `Psychic`, `Punter`, `Teenager`, `TVHost` and `Worker` by each one returning a different colour. Thus each kind of person has a different coloured face in the GUI.

```
public abstract Color getColour();
```

When the person speaks, the image is updated to show the new saying, and also is flashed twice, with a delay of 20 between each flash.

```
public void speak()
{
  latestSaying = getCurrentSaying();
  image.update();
  image.flash(2, 20);
} // speak
```

In addition to implementing `getColour()`, the `CleverPunter` class updates it image after it has made its guess for a new game.

```
public void setGame(Game requiredGame)
{
  ...
  getImage().update();
} // setGame
```

And finally, the MoodyPerson class updates its image whenever the happiness is set.

```
public void setHappy(boolean newHappiness)
{
  isHappyNow = newHappiness;
  getImage().update();
} // setHappy
```

2.5.3 The Ball class

The Ball class has a new instance variable to store a reference to its associated BallImage, and this is handled in the same way as for Person. makeImage() will be overridden in the MagicBall class that you will write, so that it returns a MagicBallImage.

```
private final BallImage image;


public Ball(int requiredValue, Color requiredColour)
{
  value = requiredValue;
  colour = requiredColour;
  image = makeImage();
} // Ball


public BallImage makeImage()
{
  return new BallImage(this);
} // makeImage


public BallImage getImage()
{
  return image;
} // getImage
```

There is also a new instance method which causes the ball to flash a given number of times with a given delay between each one. The request is simply passed on to the ball's image, which uses the SpeedController of the LotteryGUI it resides in to obtain the delays.

```
public void flash(int count, int delay)
{
  image.flash(count, delay);
} // flash
```

2.5.4 The `BallContainer` class and its subclasses

The `BallContainer` class also handles images in the same way as `Person` does.

```
private final BallContainerImage image;


public BallContainer(String requiredName, int requiredSize)
{
  name = requiredName;
  balls = new Ball[requiredSize];
  noOfBalls = 0;
  image = makeImage();
} // BallContainer


public BallContainerImage makeImage()
{
  return new BallContainerImage(this);
} // makeImage
```

The `Machine` and `Rack` classes override `makeImage()` with ones that return a `MachineImage` and a `RackImage` respectively. Notice the return type of these methods – `MachineImage` and `RackImage` are both subclasses of `BallContainerImage`.

In `Machine` we have the following.

```
public BallContainerImage makeImage()
{
  return new MachineImage(this);
} // makeImage
```

In `Rack` we have the following.

```
public BallContainerImage makeImage()
{
  return new RackImage(this);
} // makeImage
```

Back in the `BallContainer` class, when a ball is added to a ball container, the image is updated and the ball is made to flash.

```
public void addBall(Ball ball)
{
  if (noOfBalls < balls.length)
  {
    balls[noOfBalls] = ball;
    noOfBalls++;
    image.update();
    ball.flash(1, 1);
  } // if
```

```
  } // addBall
```

Similarly, when balls are swapped over they are flashed, and the image is updated when a ball is removed.

Also, in the `Machine`, just before a ball is ejected it is flashed.

```
public Ball ejectBall()
{
  if (getNoOfBalls() <= 0)
    return null;
  else
  {
    // Math.random() * getNoOfBalls yields a number
    // which is >= 0 and < number of balls.
    int ejectedBallIndex = (int) (Math.random() * getNoOfBalls());

    Ball ejectedBall = getBall(ejectedBallIndex);
    ejectedBall.flash(4, 5);

    swapBalls(ejectedBallIndex, getNoOfBalls() - 1);
    removeBall();

    return ejectedBall;
  }
} // ejectBall
```

In the `Rack` class, the sort is animated by flashing balls as they are scanned across, and just before a swap.

2.5.5 The `Game` class

The `Game` class has been changed so that a subclass of it can make subclasses of `Machine` and `Rack` if desired. The constructor calls separate methods to make the `Machine` and `Rack` instances.

```
public Game(String machineName, int machineSize,
            String rackName, int rackSize)
{
  machine = makeMachine(machineName, machineSize);
  rack = makeRack(rackName, rackSize);
} // Game


public Machine makeMachine(String machineName, int machineSize)
{
  return new Machine(machineName, machineSize);
} // makeMachine
```

```
public Rack makeRack(String rackName, int rackSize)
{
  return new Rack(rackName, rackSize);
} // makeRack
```

In the `DramaticGame` class, you will override `makeMachine()` from the `Game` class so that it makes a `DramaticMachine`.

Also the `Game` class has accessor methods to obtain the images of the `Machine` and `Rack`.

```
public BallContainerImage getMachineImage()
{
  return machine.getImage();
} // getMachineImage


public BallContainerImage getRackImage()
{
  return rack.getImage();
} // getRackImage
```

2.6 The laboratory exercise

You should work in your `$HOME/COMP16212/ex2` directory.

Copy all the java *source* files from `/opt/info/courses/COMP16212/ex2` to your directory. You should be able to do this with one `cp` command, and there is no need to copy the other files from that directory.

In addition to the usual design ideas which you write at the *start* of each part in your logbook, at the *end* of each part you must briefly record what you have done *and explain it*. This has two purposes: first to help you understand the work and remember it for the future, and second to help convince your marker that the work is your own, and you have not received too much help for it. You do not need to worry about what format you write this in, except of course that you should make some kind of a section title for each of the parts. You must write this as you proceed, otherwise you will forget your understanding and be unable to explain your work. You might also end up missing the deadline if you try to write it all at once at the end. *Your work will not be marked without the explanation.* Also, if the demonstrator marking your work feels your explanation is inadequate, or was not written at the right times, then you may be referred to me for questioning about plagiarism!

2.6.1 Warming up

When compiling and running your code, you will be using some precompiled classes that live in the jar archive file, `/opt/info/courses/COMP16212/ex2/Lottery.jar`, and so you should

write yourself the following shell scripts to help you. Write a shell script called `compile` containing the following code. Be careful to type it *exactly* correct! *DO NOT COPY/PASTE FROM PDF* – it would have non-ASCII characters!

```
#!/bin/bash
source=LotteryTest$1
TTY=$(tty)

function transitiveCompile()
{
  grep "^$1$" /dev/shm/$$-processed >/dev/null 2>&1 && return
  echo "$1" >> /dev/shm/$$-processed
  for class in $(ls *.java 2>/dev/null | cut -f1 -d ".")
  do
    grep "$class" "$1".java > /dev/null 2>&1 \
      && { transitiveCompile "$class" || return; }
  done
  test "$1".class -nt "$1".java && return
  echo Compiling "$1.java ..." > $TTY
  javac -classpath .:/opt/info/courses/COMP16212/ex2/Lottery.jar "$1".java
} # transitiveCompile

transitiveCompile "$source" > compile.out 2>&1
rm /dev/shm/$$-processed

test -s compile.out && { less -FX compile.out; exit 1; }
exit 0
```

This script will take an argument, `$1`. If that value is `A`, the script will compile the source file called `LotteryTestA.java`. If it is `B` then `LotteryTestB.java` will be compiled, and so on. With no argument, the script will compile `LotteryTest.java`. Before compiling, it first compiles any other files that are referenced from the source, if needed, and this is repeated transitively. After compiling, any error messages are stored in the file `compile.out`, which is displayed using `less` if the compilation failed.

Now write a shell script called `run` containing the following code.

```
#!/bin/bash
mainClass=LotteryTest$1
java -classpath .:/opt/info/courses/COMP16212/ex2/Lottery.jar $mainClass
```

Make both of these scripts executable. Now, by using these scripts without any argument, compile the `LotteryTest` program, and then run it. If this does not work then probably you have made an error in your scripts. Move the mouse over the image of each person and you should see their name change to `Person`. This is ready for Part A below. You can quit the program from its `File` menu (which also shows a short cut key).

2.6.2 Part A: `getClassHierarchy()` method for `Person`

Recall the new method in the `Person` class, called `getClassHierarchy()`. This simply returns the string `"Person"`. You will add a similar method to all of the 10 subclasses of `Person`. These should each return a string consisting of the name of their class, followed by `">"` then followed by the string returned by `getClassHierarchy()` from the superclass. These should *not* rely on knowing what the class hierarchy above them is, instead they must invoke the method from the superclass.

For example, the string returned by `getClassHierarchy()` in the `TraineeWorker` class should be `"TraineeWorker>Worker>MoodyPerson>Person"`.

You will write a test class, `LotteryTestA`, containing a `main()` method. This should simply create one lottery GUI, and then create one instance of each kind of instantiable person, adding it to the GUI. Your program should not make any machines, racks or balls, as they would be distractions from the test. Test programs should generally contain the minimum needed to perform the test. You also do not need to have any delays in your program.

Plan all your work for the part, including testing, in your logbook *before* commencing implementation. Remember to explain how it will work.

After implementation, compile your test program (which will also make the other classes it needs) by executing the following.

```
./compile A
```

When fixing errors, be careful to observe which class `javac` is telling you contains the error. Once these have been compiled without errors, you can run the program with the following.

```
./run A
```

Move the mouse over each person in turn. The person's name should change to their class hierarchy – check they are all correct.

If you need to correct mistakes, remember that you can compile and conditionally run in one line using the following.

```
./compile A && ./run A
```

This means "run `./compile` and if that exits with a true status, then run `./run`".

Once the part is completed you should write up your results – did it all go to plan, or did you have to make changes? Was your understanding right? Has it now improved?

2.6.3 Part B: `DramaticMachine` and `DramaticGame` classes

In this part you will improve the game by providing a dramatic lottery machine, to be used in a dramatic game. This will be the same as the existing one, except for when a ball is ejected. In an ordinary machine, a ball is picked at random, flashed, then ejected. In a dramatic machine, a ball will be picked at random, then, starting with the first ball in the machine, each ball will be flashed in turn, until the one to be ejected is reached. At that point, the chosen ball will be flashed and ejected as before.

Plan all your work for the part, including testing, in your logbook *before* commencing implementation. Remember to explain how it will work.

Create a new class `DramaticMachine` which is a subclass of `Machine`. This should override `getType()` and `ejectBall()` from the `Machine` class.

Create a new class `DramaticGame` which is a subclass of `Game`. This should override `makeMachine()` with one which returns an instance of `DramaticMachine`.

Create a test class, `LotteryTestB`, containing a `main()` method. This should simply create one lottery GUI, and then create a dramatic game, and insert it into the GUI. It should then fill up the game's machine with a number of balls (feel free to get a worker to do this – you would not need to add him or her to the GUI) and then eject a number of balls until the rack is full. You probably want to have a delay before each eject. Having any other actions (e.g. more persons, more machines) would be a distraction.

Use the same compile and run scripts as in Part A, but obviously use the argument B instead of A. Check that your machine is being labelled in the GUI as a dramatic lottery machine, rather than a lottery machine.

Explain your work in your logbook. Comment on the issue of software reuse: when overriding `ejectBall()` were you able to invoke the method from the superclass to do any of the work? Why not? What restructuring of the method in the superclass would have been useful?

2.6.4 Part C: `MagicBall` class

In this part you will further improve the game by adding a new kind of ball.

Plan all your work for the part, including testing, in your logbook *before* commencing implementation. Remember to explain how it will work.

The new class will be called `MagicBall` and it will be a subclass of `Ball`. Magic balls are just like ordinary balls, except that they also have an internal state which can change. A magic ball is always in one of 4 states as follows.

- Normal. It behaves just like an ordinary ball.
- Invisible. It cannot be seen.
- Flashing. It keeps flashing all the time.
- Counting. It keeps flashing all the time, and its observable value keeps changing each time it flashes. The value appears to count from 0 to 99 and round again. The actual value of the ball does not change however, only the value returned by the `getValue()` method.

To implement the counting state, you will need to override `getValue()` from `Ball`. Hints: You will need an additional instance variable to store the last value returned by `getValue()`, so that it can be changed each time, when the ball is in the changing state. You will need to access `getValue()` from the superclass when the ball is not in the changing state.

I have provided the class `MagicBallImage` to support magic balls, including making them disappear and continually flash (using a separate thread) when required. In `MagicBall`, you will need to override `makeImage()` from `Ball` to ensure that magic balls are shown using an instance of `MagicBallImage` rather than `BallImage`.

The different natures of a magic ball are probably best modelled as 4 states, one for when it is behaving like a normal ball, one for when it is not visible, one for when it is flashing but not changing its value, and one for when it is flashing and is changing its value. You should decide how to represent these 4 states (hint: you have seen similar things in the snake case study). When a magic ball is created, it should behave like a normal ball.

In order to interact properly with its image object, a magic ball will also need to provide the following methods:

Method interfaces for class `MagicBall`.

Method	Return	Arguments	Description
`doMagic`	`void`	`int spellNumber`	When `spellNumber` has the value 1, this will make the state of the magic ball change to the next state; going back to the first state after the last. When `spellNumber` has the value 2, this will make the state of the magic ball change to its normal state. If `spellNumber` has any other number, it will do nothing, or it might perform a *different* action: this is available for feature extensions.
`isVisible`	`boolean`		Returns `true` if the magic ball is in a state which requires the image to be visible, `false` otherwise.
`isFlashing`	`boolean`		Returns `true` if the magic ball is in a state which requires the image to flash, `false` otherwise.

These methods can be used from anywhere in the program (e.g. from `main()`), but are primarily designed to interact with the magic ball image. If the program user clicks on the image of a magic ball, the magic ball image will invoke the `doMagic()` method. The `spellNumber` will be 1, if the left button is clicked, 2 for the middle, and 3 for the right button.

Don't forget that each time a magic ball changes state, it must update its image so that the change is reflected in the GUI. This is done by calling the `update()` method of its image object.

Create a test class, `LotteryTestC`, containing a `main()` method. This should simply create one lottery GUI, and then create a game with a small machine and a rack, and insert them into the GUI. It should then insert a small number of magic balls into the machine. These can then be tested by left clicking on them.

Once the part is completed you should write up your results – did it all go to plan, or did you

have to make changes? Was your understanding right? Has it now improved?

2.6.5 Part D: `MagicWorker` class

Plan all your work for the part, including testing, in your logbook *before* commencing implementation. Remember to explain how it will work.

In this part you will improve the game by providing a magic worker. This will be the same as a worker, except he or she will always create magic balls whenever he or she is asked to make balls.

In addition, a magic worker will secretly keep track of all the balls he or she has ever made, by storing a reference to them in an array. You will need to extend this array if it gets full and another ball is made. (You may instead read ahead and use an `ArrayList`.) These references can then be used to support the following extra method.

Method interfaces for class `MagicWorker`.

Method	Return	Arguments	Description
`doMagic`	`void`	`int spellNumber`	This causes the `doMagic()` method of every magic ball made by *this* worker, to be invoked with the same `spellNumber` as given here.

To support magic workers, I have provided a `MagicWorkerImage` class, a subclass of `PersonImage`. You will need to override the `makeImage()` method of the `Person` class in order for a `MagicWorker` to have an image from the `MagicWorkerImage` class.

A `MagicWorkerImage` causes the `doMagic()` method of the associated `MagicWorker` to be invoked whenever the program user clicks on the image. It passes the value 1 if the left button is clicked, 2 for the middle, or 3 for the right.

Create a test class, `LotteryTestD`, containing a `main()` method. This should simply create one lottery GUI, and then create two magic workers, and insert them into the GUI. Then it should create two games and insert them into the GUI. It should then cause one magic worker to fill one machine, and the other magic worker to fill the other. Next it should eject some balls from each machine, inserting them into the corresponding racks.

If you have used an array to store the balls created, you should make sure the code to extend the array is tested in at least one of the magic workers. So, make the array small to start with.

The magic workers can now be tested by clicking on them: they should each do magic on all, and only, the balls that they created.

Once the part is completed you should write up your results – did it all go to plan, or did you have to make changes? Was your understanding right? Has it now improved?

2.6.6 Part E: Extensions

If (and only if) you have time, there are many extensions you could try. To help your imagination, some suggestions are as follows. These are mainly for fun, and so there are only a few marks allocated to them. However, if you found the previous parts too easy then you might learn something here.

1. E1 A magic ball could be told which ball was created before it, and which was created after it. Then, when given spell number 3, it could change state but also pass the same spell to the previous and next balls, thus resulting in instant chaos!
2. E2 (Tricky?) Whenever a magic ball is put into or taken out of a dramatic machine, the ball could be informed, and given a reference to the machine (or null when it is leaving). This could then enable a magic ball to find its own position inside the machine, and swap its position with another ball in the machine. This could be an alternative use for spell number 3, so clicking on it with the right button would cause it to move away! Note: a dramatic machine must be able to cope with any type of ball, but only magic balls could be told what machine they are in, so you would need to use `instanceof` and some casting.
3. E3 (Hard?) You could add a new state to magic balls, in which they share their value with all other balls in the same state. Each time a ball goes into or out of the sharing state, all balls in the sharing state immediately appear to have the same value. This value is the mean average of the actual values of all those balls (rounded to a whole number). To make it trickier, the balls should not flash when in the sharing state. Note: the balls in the sharing state may have been made by different magic workers, or not have been made by any magic worker.

Create a test class, `LotteryTestE`, containing a `main()` method, to test your extensions. For ease of marking (etc.) create only one test class. However, if you have more than one extension, then each should be tested in a separate static method, and the test which the program carries out should be selected via a command line argument. You could also use that argument to tell the `MagicBall` class what the meaning of spell number 3 is.

As before, don't forget to write up and explain your work in your logbook: planning in advance, results afterwards.

2.7 Assessment criteria

The exercise is marked out of 60 marks, and is worth 12.5% of the assessment for the COMP16212 laboratory.

The exercise will be marked as follows. When marking the pieces of code, the completeness and correctness will be judged, and then the quality will be considered for completed pieces.

Part	Aspect	Marks		
		Correct	Quality	Other

Part	Aspect	Marks		
		Correct	Quality	Other
A	`getClassHierarchy()` method in 10 subclasses	4	4	
A	Appropriate use of logbook			2
A	Total	10		
B	Constructor for `DramaticMachine`	1	1	
B	`getType()` method	1	1	
B	`ejectBall()` method	3	3	
B	Appropriate use of logbook			5
B	Total	15		
C	Modelling of states for `MagicBall`	1	1	
C	Constructor for `MagicBall`	1	1	
C	`makeImage()` method	1	1	
C	`getValue()` method	1	1	
C	`doMagic()` method	1	1	
C	`isVisible()` and `isFlashing()` methods	1	1	
C	Appropriate use of logbook			3
C	Total	15		
D	Constructor for `MagicWorker`	1	1	
D	`makeImage()` method	1	1	
D	`getPersonType()` method	1	1	
D	`getClassHierarchy()` method	1	1	
D	`makeNewBall()` method	1	1	
D	`doMagic()` method	1	1	
D	Appropriate use of logbook			3
D	Total	15		
E	E1	1	1	
E	E2	1	1	
E	E3	1	1	
E	Others – on merit, but not easy marks			
E	Appropriate use of logbook			1
E	Total (*maximum for Part E*)	5		
	Maximum marks	60		

2.8 Completing your work

As soon as you have completed, use `submit` to submit your work and then `labprint` to produce a listing ready for marking. The required files are named as follows.

```
compile
run

AudienceMember.java
CleverPunter.java
Director.java
MoodyPerson.java
Psychic.java
Punter.java
Teenager.java
TraineeWorker.java
TVHost.java
Worker.java
LotteryTestA.java

DramaticGame.java
DramaticMachine.java
LotteryTestB.java

MagicBall.java
LotteryTestC.java

MagicWorker.java
LotteryTestD.java

LotteryTestE.java
```

COMP16212 – Exercise 11o (0 Sessions – optional) Ordered Binary Trees

(and also Shell Scripts and LaTeX thrown in for no extra charge!)

John Latham

11o.1 Aims

The main aim of this exercise is to make you focus on recursive data types, in particular ordered binary trees, through you creating a class which is similar to the one you have seen demonstrated in the lectures. This work will turn the insights you gained through seeing those demonstrations, into a concrete understanding – in readiness for the examination if nothing else!

A secondary aim is to continue to expose you to a wider issue, that real world programs do not stand alone. They are integrated with other, often standard, programs, to make systems. The Unix world is particularly suited to this. The exposure is achieved here by inviting you to look at the interaction between your java programs and various shell scripts that manipulate them.

A third aim is to encourage you to take another look at the powerful LaTeX document processing system.

11o.2 Learning outcomes

On successful completion of this exercise, a student will:-

- Have used (again) an interface defined in the standard Java API.
- Have had some experience of writing a recursive datatype, and code to manipulate it.
- Have had more experience of writing a generic class.
- Have had more exposure to shell scripts which can be used for testing, and for constructing simple applications.
- Have has another opportunity to take up the invitation to study a simple introduction to the powerful LaTeX document processing system.

11o.3 Overview

In the lectures we have studied ordered binary trees of integers greater than or equal to 0. In reality we would like our trees to be able to contain any kind of object that has a total order. You

have met the interface called `Comparable` in the package `java.lang`. You may find it helpful to take another look at it in the on-line documentation. (Surely you have bookmarked the API specification in your browser by now?)

You will be writing a generic class which offers ordered binary trees of items which implement the `Comparable` interface. Your OBT class will be called `OBTComparable`.

You may need to review the lecture notes and/or the chapters on interfaces and generics.

11o.4 Exercise parts

The exercise is divided into 6 parts, A to F. You are not expected to complete or even attempt all of these.

Part A asks you to implement and test the class `OBTComparable` with methods `insert()` and `find()`. Then write a simple test program for it. I expect all of you to complete this part.

Part B asks you to add the `getSize()` and `getDepth()` methods. Then write a simple test program which integrates with a shell script I have provided. I expect most of you to complete this part.

Part C asks you to study the shell script used to test part B. You do not have to become a complete expert in shell scripts, merely expand your knowledge by reading it seriously, attempting to understand it, and be able to convince your marker that you have done this. I expect most of you to complete this part.

Part D asks you to add the `elementsAscending()` method to `OBTComparable` and write a program which sorts its input using a tree sort. You will then use this in a simple shell script I have provided which implements a crude spelling checker. I expect about half of you to complete this part.

Part E is an extra, and asks you to write the whole spelling checker in Java. I expect only a few of you to complete this part.

Part F is an extra extra, and asks you to write a better spelling checker in Java with lots of flashy features! I expect hardly any of you to complete this part.

Part	Description	Marks
A	`insert()` and `find()`	20
B	`getSize()` and `getDepth()`	20
C	Study of shell script	10
D	`elementsAscending()` and sort	10
E	Spelling checker	5
F	Fancy spelling checker	5
Logbook	Appropriate and sensible use of logbook throughout	5
	Total	75

11o.5 The laboratory exercise

You should work in your `$HOME/COMP16212/ex11o` directory.

Copy all the files from `/opt/info/courses/COMP16212/ex11o` to your `$HOME/COMP16212/ex11o` directory.

```
cd ~/COMP16212/ex11o
cp /opt/info/courses/COMP16212/ex11o/* .
```

Now look at each file using `less`.

`depth-size-test` This is a shell script that you will use to experiment with the relationship between size and depth of a randomly created tree; compared with a balanced one. It will draw a graph for you. You will use it in part B and study it in part C.

`latex-intro.tex` This is the source of a LaTeX document which you will use in part D as input to your spelling checker. It is also a simple introduction to LaTeX, and you are strongly encouraged to read it straight after completing the exercise.

`run-me-to-view-latex-intro` This is a simple shell script to help you read the above LaTeX document.

`spell-check` This is a shell script that uses the sort program you are going to write, to provide a crude but usable spelling checker. You will use it in part D.

11o.5.1 Part A: `OBTComparable` with `insert()` and `find()`

Write the `OBTComparable` class including the methods `insert()` and `find()`. Your implementation will probably be similar to the class `OBTInt` you have seen in the lectures, except that the items in the tree can be objects of any class which implements the `Comparable` interface. `OBTComparable` should be a generic class, with a single type parameter denoting the type of the elements in the tree. The type must be `Comparable` with itself. You will need to think carefully where to use the type parameter within the class itself.

Write a test program in a separate class called `FindTest`. This should insert a number of words of type `String` into a tree of strings and then search for a number of strings, some of which should be found and some not. It should report on standard output, using `System.out.println()`, the strings which are inserted into the tree and those which are searched for and whether they are found.

Note that `String` implements `Comparable<String>`.

Your insertion test data should be a sentence describing who you are and what your hobbies are (make the hobbies up if you must!), as in: `My name is John Latham and my hobbies include films music photography electronics and computing`. You do not want any punctuation, just words. However, you should have your words capitalised as appropriate.

Your search data should be a list of hobbies, some of which are yours and some which are not. You should also include your name twice, once with capitalisation and once without.

My example output is as follows.

```
Inserting My
Inserting name
Inserting is
Inserting John
Inserting Latham
Inserting and
Inserting my
Inserting hobbies
Inserting include
Inserting films
Inserting music
Inserting electronics
Inserting and
Inserting computing
Searching for latham: false
Searching for films: true
Searching for swimming: false
Searching for fishing: false
Searching for computing: true
Searching for climbing: false
Searching for paragliding: false
Searching for Latham: true
```

Your test data should be built in to your main method, but stored in two arrays for ease of processing and maintenance. One array will contain the strings to be inserted in the tree and the other will house those to be searched for. For example:

```
String [] insertStrings
  = new String [] { "My", "name", "is", "John", "Latham", "and", "my",
                    "hobbies", "include", "films", "music", "electronics",
                    "and", "computing" };
```

These can then be processed using a loop for each.

Store the output from your program in a file called `find-test-results`.

```
java FindTest > find-test-results
```

11o.5.2 Part B: adding `getSize()` and `getDepth()`

Add the methods `getSize()` and `getDepth()` to your `OBTComparable` class.

Write a test program in a separate class called `DepthSizeTest`. This should create an empty tree of `String` and then read strings from the standard input, one line at a time, and insert them into the tree. After each insert, the program should report one line on the standard output, containing just two numbers: the size of the tree and its depth, with a single space between. *This format is important in order for the test script I have provided to work.* The program should

end when there is no more input. You can implement all this directly in the main method of `DepthSizeTest`.

To remind you from your study of files, here is a simple way to read lines of text, each as a string, from the standard input.

```
// create a BufferedReader for the standard input
BufferedReader bufferedReader
   = new BufferedReader(new InputStreamReader(System.in));
try
{
  String line;
  // read a line of text from the input until there is no more
  while ((line = bufferedReader.readLine()) != null)
  {
    // do what you want to with line
  } // while
} catch (IOException e) { System.out.println(e); };
```

Run the program and type some strings into it, one per line. This should cause output to appear after each one. Make sure there are two numbers on each line, the size of the tree, followed by one space, followed by the depth of the tree, and nothing else. You do remember how to signify the end of file on standard input, don't you? (control-D)

Once you are convinced the program is working, run the script `depth-size-test`. This should run the `DepthSizeTest` program, pass 100 words chosen at random from the on-line dictionary to its input, and take its output as part of the input to a graph plotting program called `gnuplot`. The script also simulates size and depth data for balanced trees of the same sizes and passes both graphs to gnuplot. The result should be a gnuplot window showing graphs of your experiment compared with the minimal depths possible.

11o.5.2.1 Obtaining hard copy

Run `depth-size-test` several times. You should get different but similar results. Obtain hard copies of two of these graphs. The easiest way to obtain a hard copy is to use the menu option `Start menu -> System Utitlities -> Capture (Print) Window` in AnotherLevelUp. This pops up a form which allows you to **Capture** a window's image and then preview it and send it straight to the printer. Press `Capture` and then click in the window whose image you want. Make sure the window you want is completely on top when you capture it, or you'll get it complete with parts of other windows hiding it! (The capture form will hide itself during the capture so it does not matter if that is obscuring part of your window.) Press `Preview` to check this image. You may have to change the name of the printer in the form. Then press `Print` to send it to the printer. The image should get auto sized and rotated if necessary to make it fit nicely on the paper.

Note: if the printers are busy then the above method to produce a hard copy might take too long. This is because it generates a bit map image of the window and converts it to postscript – resulting in a huge printer file. When you look at the script `depth-size-test` in the next

section, you will see an alternative way of getting a hard copy. Although less convenient it uses a directly generated postscript file which is therefore much smaller than a bit map.

11o.5.2.2 More experiments

Look at the script `depth-size-test` (using `less`: you are not going to edit it, so why use your editor just to look at it, when this takes longer and also runs the risk of accidentally changing it?!)

Find out how to have the words sorted before they are passed to your java program, and also how to change the number of words used in the experiment.

Obtain four more hard copies of the graphs resulting from:

- 1000 words which have not been sorted.
- 100 words which have been sorted.
- 1000 words which have been sorted.
- 10000 words which have not been sorted (this one will take a while to extract 10000 words from the dictionary).

You should now have a total of six graphs, which you will later attach to your `labprint` output before marking.

11o.5.3 Part C: a brief study of a shell script

Most real systems in the Unix world include shell scripts as well as programs written in programming languages. Whilst it is not directly programing in Java, it is important for you to appreciate the power available when you combine your programs with the standard tools of Unix. Not just for testing but possibly as an integral part of the system.

You have just used the script `depth-test` to use your Java program in an experimental exploration of the nature of OBTs. The script combined algorithms of its own (e.g. to produce the optimal graph data) with feeding data through your program, ultimately into a powerful graph plotting program, called `gnuplot`.

You should already be interested to know how it works. But as an extra incentive, you can have some easy marks for trying to find out! All you have to do is read the script and attempt to understand it. There are comments in it explaining all the concepts used, so you don't even have to read the `bash` man page. It doesn't matter if you do not understand every part of it deeply, but you are looking for a basic grasp of its structure and the main features used. This should serve as a continuation of your introduction to shell scripts, so that you might feel more inclined to write scripts of your own when appropriate – perhaps much simpler ones at first.

You will be asked to explain the script during the marking process. I recommend you write some notes about your understanding in your logbook. What new things have you learnt about shell scripts from this process?

11o.5.4 Part D: adding `elementsAscending()`

Add the method `elementsAscending()` to `OBTComparable` to produce an ascending in-order list of the elements. Your method should return an `Iterator`. (Note that the example in the lecture notes wrongly used the raw-type `ArrayList`. You should supply an appropriate type parameter in your code.)

Write a program in a separate class called `Sort`. This should take a list of strings from the standard input, one per line, and produce an ascending sorted version of the list, still one string per line, on the standard output. It should use **tree sort** to obtain the result.

To test your program, you have been provided with another shell script called `spell-check`. This uses your sort program and others to make a crude spelling checker. Text is taken from standard input, divided into words, sorted by your program, and then compared with an on-line dictionary. Those words from the input which are not found in the dictionary are placed on standard output. Run the script taking input from the file `latex-intro.tex`.

```
./spell-check < latex-intro.tex
```

If all goes well, the output should consist of 18 unrecognised words, the first five of which are `CONTENTS GENERALLY IMPAIRED WITH WYSIWYG`. You can check the number of words as follows.

```
./spell-check < latex-intro.tex | cat -n
```

11o.5.5 Part E: (extra) making a spelling checker

Make a program in a class called `SpellCheck1` which has the same functionality as the script `spell-check` and works in a similar way: filter out unwanted characters and latex commands, sort the remaining words, and sequentially compare with the dictionary. You will need to look at the `spell-check` script in order to see its full functionality.

You should ensure all your Java code is in the file `SpellCheck1.java` (even if you have more than one class in that file).

Ultimately your new program should all be written in Java, but you might do it in stages by making a copy of `spell-check` and gradually moving some functionality from it into `SpellCheck1.java`. E.g. you might implement the `uniq` and the `comm` functionality first, whilst leaving the input filtering code in your copy script.

Try your program using the `latex-intro.tex` source file. You can check to see if it produces the same output as follows.

```
./spell-check < latex-intro.tex > output-from-spell-check
java SpellCheck1 < latex-intro.tex > output-from-SpellCheck1
diff output-from-spell-check output-from-SpellCheck1
```

If they are identical, the `diff` program will produce no output.

What observations have you made during the process of converting `spell-check` into `SpellCheck1`?

11o.5.6 Part F: (extra, extra) making a better spell checker

Make a program in a class called `SpellCheck2`. This should build a tree from the *dictionary* file knowing that it is already sorted, and so efficiently build a *balanced* OBT. This will require you to add to the class `OBTComparable` the method `buildFromList()`. You will need to read the file twice, once to count the words and once to build the balanced tree.

Having built the dictionary tree, your program should then take the document from the standard input, perform the same filtering on it as for `SpellCheck1`, to get words which need to be spelling checked. These should then be searched for in the tree, and if not found reported somehow on the output.

It is acceptable to copy and paste the input filtering code from `SpellCheck1` to `SpellCheck2`, to avoid the need for a shared source file the name of which is not known to `labprint`.

Of the two programs, `SpellCheck1` and `SpellCheck2`, which approach is best, and why?

Now you can use your imagination. Offer the user various command line options to affect the output, such as:

- displaying just the words not found, once each, sorted.
- displaying each word not found together with its line and word number from the original document, in position order.
- producing a new version of the document with mis-spelled words displayed annotated, perhaps by placing three question marks either side of them.

11o.6 Assessment criteria

The exercise is marked out of 75 marks, and is worth one **optional session** of the assessment for the COMP16212 laboratory.

The exercise will be marked as follows.

Part	Aspect	Marks
A	Basic `OBTComparable` correctness and quality	4
A	`insert()` correctness and quality	4
A	`find()` correctness and quality	4
A	`FindTest` class	6
A	File `find-test-results` exists with output results in it	2
A	Total	20
B	`getSize()` correctness and quality	4
B	`getDepth()` correctness and quality	4
B	`DepthSizeTest` class	6
B	6 graph printouts correct	6
B	Total	20

Part	Aspect	Marks
C	Has studied `depth-size-test` script seriously	6
C	Level of understanding	4
C	Total	10
D	`elementsAscending()` correctness and quality	4
D	`Sort` class	6
D	Total	10
E	Spelling checker correctness	4
E	Spelling checker quality	1
E	Total	5
F	Fancy spelling checker appropriateness and correctness	4
F	Fancy spelling checker quality	1
F	Total	5
Logbook	Used appropriately and sensibly throughout the exercise	5
	Maximum marks	75

11o.7 Completing your work

As soon as you have completed, use `submit` to submit your work and then `labprint` to produce a listing ready for marking. The required files are named as follows.

```
OBTComparable.java

FindTest.java
find-test-results

DepthSizeTest.java

Sort.java

SpellCheck1.java
output-from-spell-check
output-from-SpellCheck1

SpellCheck2.java
```

You should also attach your 6 graphs to your `labprint` document prior to marking.

COMP16212 – Exercise 12o (0 Sessions – optional) Flight Booking: A GUI for a Flight Booking System

Sean Bechhofer

12o.1 Aims

This laboratory will enable you to consolidate your understanding of Graphical Interfaces and the Java Swing Libraries.

12o.2 Learning outcomes

On successful completion of this exercise, a student will:-

- have taken a skeletal user interface and user requirements, and designed functionality for the user interface;
- have implemented such functionality using appropriate methods from the Java Swing Libraries;
- have maintained design modularity: providing a user interface based on a given implementation of a model without altering the model's interface.

12o.3 Introduction

The general task is to implement a Graphical User Interface (GUI) allowing the user to query and book flights. You will be provided with a basic skeleton of the GUI – all the components such as buttons, lists and menus will already be added to the interface. The given interface has no functionality though. Pressing buttons will have no effect. Your task is to implement appropriate methods that provide the required functionality of the interface, and to connect those methods with the interface components through an appropriate use of `Listener` objects.

You will be given an implementation of the *model* used in the application, e.g. the classes that represent the basic objects being manipulated. You will also be given a main class that will drive the application. Thus, before you have done any work, you should be able to see what the application looks like (although it will not *do* anything. This will also ensure that you can test your code and see the results of changes *as you go*.

As with earlier exercises, you will need to think about the design of your application away from the computer before coding.

12o.4 Stages of development

The work is divided into seven parts. It is *not* expected that everyone will complete all the parts.

We expect that all of you will get to the end of Part C. Over half of you will finish Part E, a quarter of you will get on to Part F, and only a handful will attempt Part G.

Make sure that you read through the *entire* lab description before starting work – the requirements for later parts may impact on your design of the earlier sections.

12o.4.1 Deadlines

There is one session assigned to this lab.

12o.5 The Model

The lab is based around a scenario of booking flights, and a model is provided with objects representing flights and flight bookings. The model is described briefly here, but you should also make use of the supplied `javadoc` in order to understand the code, what it does and how to use it. Note that for this exercise we do *not* supply you with the source code for the model – you have to rely entirely on the descriptions of the public methods in the class and the `javadoc`.

There are three basic classes provided: `Flight`, `FlightBooking` and `FlightBookingManager`. These are actually defined as Java *interfaces*, with a class `FlightBookingManagerImpl` providing an implementation. In addition, a `Main` class drives the application.

`Flight` Represents a flight. Each flight has a departure airport, a destination airport, a flight code and a date. The properties of a flight are all represented by `String` values[1]. All flights are one-way (single).

`FlightBooking` Represents a flight booking. A booking is for a particular flight and holds information about the passenger making the booking, the class[2] of the booking and credit card details. The passenger details consist of a title and name. A booking also has a reference number. Special requests can be added to a booking.

The `FlightBooking` class also provides some constants (`public static final` variables) that are used when creating bookings. These include constants for the various booking classes and an array of strings that are allowed in titles of passengers.

`FlightBookingManager` Represents a booking system. The Booking Manager can be queried for available flights between two airports on a given date. Bookings can also be made on particular flights. Bookings can be cancelled, and all the current bookings can

[1] Flight dates could have been represented using the `java.util.Date` class. However, this would then require conversion of string values to dates in the user interface. To simplify the example, `String` is used. The implementation ofthe classes may, however, perform some validation on the values passed in when querying flights to make sure that they are “sensible” date values.

[2] By class here we mean first/business/economy rather than Java class!

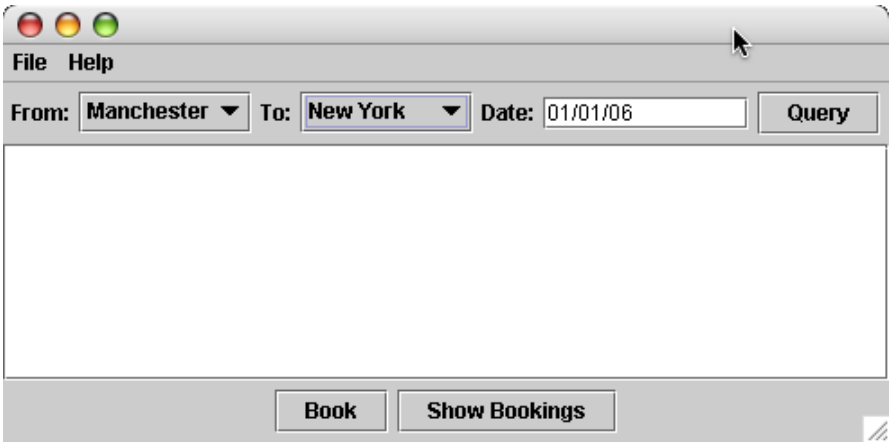


Figure 1: Main Application window

be retrieved. Exceptions may be thrown by some of these methods (see the `javadoc` for details). For example, departure and destination aiports in a query must be different, and there are restrictions on what constitutes a valid credit card number for a booking.

The Booking manager supplies a list of airports that it can book flights between.

`FlightBookingManagerImpl` Provides an implementation of `FlightBookingManager`.

`Main` Provides a `main()` method that creates a `FlightBookingManagerImpl` and opens up a `FlightBookingManagerGUI`. The application is run using this class, e.g.

```
java <arguments setting classpath> Main
```

12o.6 The User Interface

This lab is about producing a Graphical User Interface (GUI) that allows a user to query and book flights. The basic GUI is given – your task is to implement appropriate methods that supply the required functionality. In the following section, we describe the expected functionality of the interface. Note that the order in which functionality is described here is not necessarily the order in which we expect you to implement it. Section 12o.7 describes the order in which you should approach the implementation tasks.

Figure 1 shows the main application window.

12o.6.1 Save and Load

Collections of bookings can be saved and loaded in files[3]. The **File** menu should provide access to this and via the following three menu items:

[3]The details of the save and load format are not important to you in this exercise – you should only be concerned with the fact that save and load are possible. In actual fact, the bookings are stored as binary files.

Load Bookings... Opens a file chooser allowing the user to select a file of previously stored bookings. Bookings are held in files with the extension `.bkg`.

Store Bookings... Opens a file chooser allowing the user to select a file into which the current bookings are saved. Note that the `storeBookings` method will *overwrite* the contents of the file if it already exists.

Quit Close the application. A pop up should ask the user to confirm that they really want to exit.

A **Help** menu should be provided, with a single item **FlightBooker Help**. When selected, a pop up window should appear, with basic instructions for the tool.

The application can also be closed down by closing the window using the "standard" window buttons. As with the **Quit** button, a pop up should ask the user to confirm that they really want to exit.

12o.6.2 Querying Flights

When the **Query** button is pressed, the flight booking manager should be queried for flights from `departure airport` to `destination airport` on `date`, where the airports are given by the values selected in the appropriate pull down boxes and the date is the given by the value in the text field.

The flights retrieved should then be displayed in the list.

12o.6.3 Booking Flights

If a flight is selected in the list of flights, and the **Book** button is pressed, a new window should open up allowing the user to book a seat on the selected flight.

Figure 2 shows the booking window open for a particular flight. Details can be added here such as passenger name, required booking class and credit card number.

If the **OK** button is pressed, an attempt to book the flight (using the given information) should be made. If successful, the window should close. If unsuccessful for any reason, the window should not close and the user should be informed of the problem via a pop up window.

If the **Cancel** button is pressed, the window should close without the booking being made.

The window *cannot* be closed using the "standard" window buttons, but *must* be closed using either the **OK** or **Cancel** buttons.

12o.6.4 Querying Bookings

If the **Show Bookings** button is pressed, a window should open up showing all the bookings for the Booking Manager.

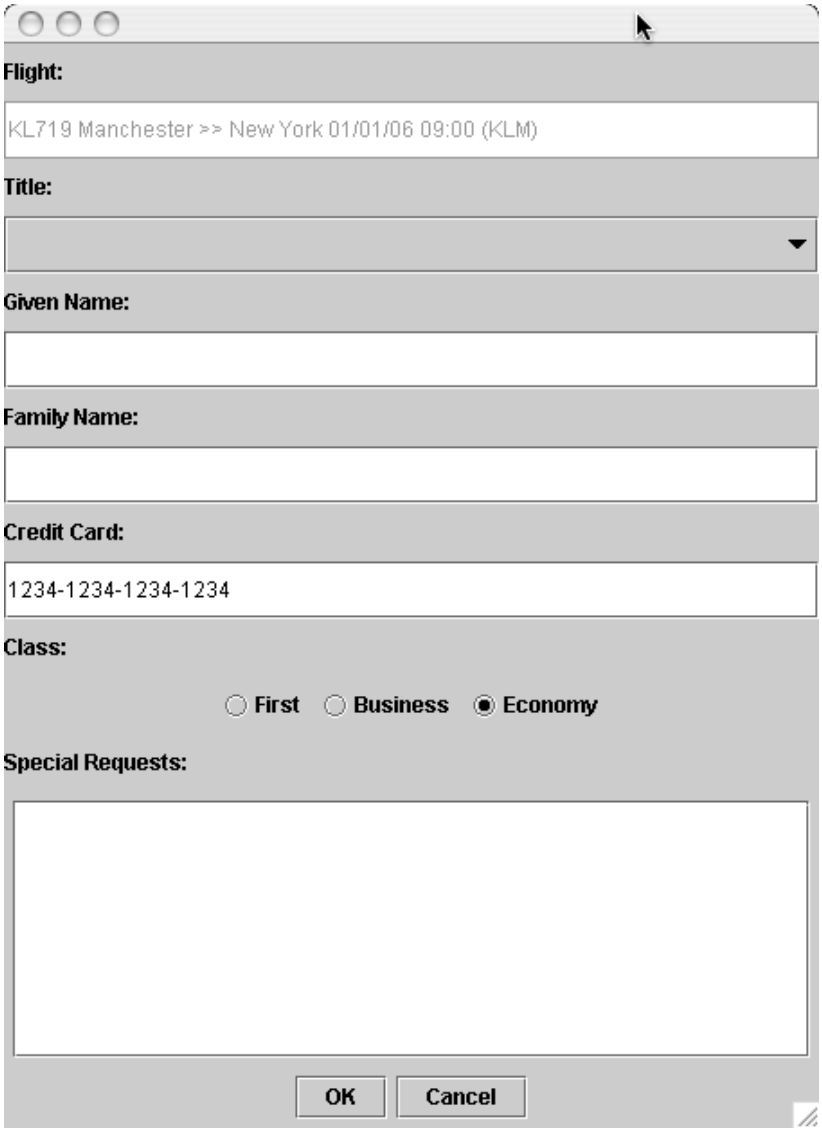


Figure 2: Booking Window

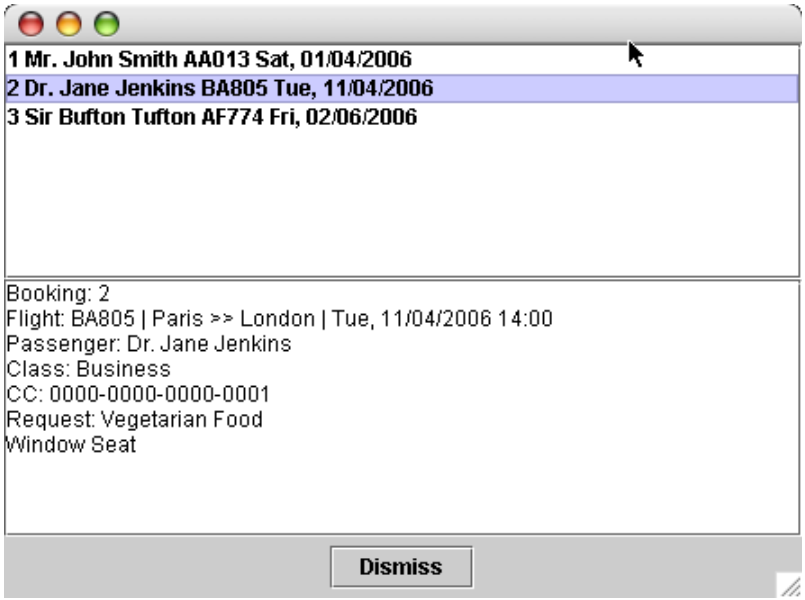


Figure 3: Bookings Window

Figure 3 shows the bookings window. All bookings are shown in a list in the top half of the window. If a booking is selected, the details of the bookings are shown in the text area in the bottom half of the window.

Pressing the **Dismiss** button will close the booking list window.

The window *cannot* be closed using the "standard" window buttons, but *must* be closed using the **Dismiss** button.

12o.7 The laboratory exercise

You should work in your `$HOME/COMP16212/ex12o` directory. A number of files are supplied for you. These can be found in directory:

```
/opt/info/courses/COMP16212/ex12o
```

12o.7.1 Class Files

Java class files for the basic model are supplied in a `jar` file
`/opt/info/courses/COMP16212/ex12o/flights.jar`

When compiling and running your code, you will need to make sure that the jar file is on your classpath. It might help you to write some shell scripts that run `java` and `javac` with appropriate arguments (you will have already done this for exercise 1).

12o.7.2 Javadoc

Documentation for the classes (generated using `javadoc`) is in directory `/opt/info/courses/COMP16212/ex12o/javadoc`. You should *regularly* be consulting this documentation while developing your solution.

12o.7.3 Source Files

You will need the following source files:

```
BookFlightWindow.java
FlightBookingManagerGUI.java
ShowBookingsWindow.java
Main.java
```

Make copies of these files in your `$HOME/COMP16212/ex12o` directory.

During the course of the exercise, you will edit the source files, adding methods and altering the existing code.

Before you start, compile and run the given code in order to check that everything is ok and any shell scripts you may have written are working.

Note that throughout the exercise, where listeners are called for (e.g. on the **Query** or **Book** buttons) you should use *anonymous* listeners.

HINT: The descriptions of the tasks may include hints, for example suggesting the classes that you probably need to use.

12o.7.4 Data

A collection of bookings `/opt/info/courses/COMP16212/ex12o/bookings.bkg` is provided for you to test operations loading bookings. Copy this to your working directory.

12o.7.5 Part A: Component Initialisation

Alter the initialisation of the `JComboBoxes` in `FlightBookingManagerGUI` so that they show the airports that the `FlightBookingManager` can provide flights for.

HINT: Setting `JComboBox` contents

12o.7.6 Part B: Window Closing

Add a listener that will check with the user (via a pop up window) when they attempt to close the window using the window controls.

HINT: Using `WindowListener`, `JOptionPane`

Add a menu item to the **File** menu that allows the user to quit the application. Ensure that the user is asked to confirm if this is what they want to do.

HINT: Using `JMenuItem`, `JOptionPane`

12o.7.7 Part C: Query

Implement the required functionality for querying flights. You should do this by adding an anonymous listener to the **Query** button. You may also need to add methods to the `FlightBookingManagerGUI` class that implement the query. The contents of the `JList` showing flights in the interface should be set to be the results of querying the `FlightBookingManager` for appropriate flights. The parameters for the query should be obtained from the airport selectors and date entry field.

Ensure that any exceptions are handled appropriately.

HINT: Using anonymous listeners, getting values from `JComboBox`, setting `JList` contents, exception handling.

12o.7.8 Part D: Booking a Flight

Implement the required functionality for booking flights, i.e. opening up a `BookFlightWindow` window that allows entry of data for a particular booking. Use an anonymous listener added to the **Book** button.

HINT: Using an anonymous listener, opening a new window, exception handling, `JTextArea`, dismissing windows.

Ensure that only one booking class can be selected. The `JComboBox` for titles should be initialised with appropriate values taken from the `FlightBooking` class.

HINT: Using `JRadioButton`

Ensure that the window closing behaviour is as specified in the description in Section 12o.6.

HINT: Setting `defaultCloseOperation`

12o.7.9 Part E: Displaying Bookings

Implement the required functionality for displaying the current bookings, i.e. open up a new `ShowBookings` window. The list in the upper half of the window should show all the current bookings. When one is selected, the details of the booking[4] should be shown in the text area in the lower half of the window. Use a `MouseListener` to detect when the user has selected an item.

HINT: Setting `JList` contents. `MouseListener` on `JList`.

[4] Hint: The method `getDetailedDescription()` on class `FlightBooking` is likely to prove useful here.

Ensure that the window closing behaviour is as specified in the description in Section 12o.6.

12o.7.10 Part F: Save and Load

Add menu items to the main menu in `FlightBookingManagerGUI` that will save the current collection of bookings to a file and load bookings from a file. Use a `JFileChooser` to allow the user to select the file.

Ensure that any exceptions are handled in a sensible way.

HINT: Using `JMenuItem`, `JFileChooser`, exception handling.

12o.7.11 Part G: Optional Extras

There are a number of enhancements you can consider making to the application. There are *no marks* awarded for these – make sure that you have completed everything in parts A–F before you consider these.

- When saving, if the file already exists, get the user to confirm that they are happy overwriting the file.
- When saving, make a backup of the file in `<file>.bu`, where `<file>` is the name of the existing file.
- Add colour to the interface.
- Add the ability to *delete* bookings in `ShowBookingsWindow`. Note that the functionality for deleting bookings is already present in the model. Think carefully about issues that may be brought up by deleting bookings.
- Improve the layout. In particular, improve the layout used in `BookFlightWindow`. Although it has not been covered in the lectures, you may find that `GridBagLayout` proves useful here.
- Add the possibility of booking return flights.
- Using a `MouseListener` for the `ShowBookings` window has a drawback. Can you work out what it is? (Hint: think about what the arrow/cursor keys do). How might you address this problem? Answering this may require you to investigate other listener classes that have not been introduced in the lectures.
- What changes to the code might be required in order to allow the user to *change* bookings, e.g. move a flight to the next day? Would it be possible to do this with the classes given? Or would you need to change the internal implementation?
- The sample code here uses classes to provide the implementation of the model. In the diary example used in the lectures, a Java `interface` is used to describe how we access the model. How might this approach be used here? What changes would it make to the implementation. We do not expect you to change the implementation here, but you should write down some thoughts in your log book.

12o.8 Assessment criteria

The exercise is marked out of 100 marks, and is worth one **optional session** of the assessment for the COMP16212 laboratory.

The exercise will be marked as follows.

Part	Aspect	Marks		
		Correctness	Quality	Other
A	Off-line Design and write up in logbook			2
A	Accessing data	2		
A	Initialising components	4		
A	Overal Quality		2	
A	Total	10		
B	Off-line design and logbook writeup			2
B	Window Listener	5		
B	Quit Menu Item	5		
B	Popup Menu	4		
B	Overall quality and test		4	
B	Total	20		
C	Off-line design and logbook writeup			2
C	Button Listener	5		
C	Query	5		
C	Setting list contents	5		
C	Exception Handling	5		
C	Overall quality and test		3	
C	Total	25		
D	Off-line design and logbook writeup			2
D	Opening Window	4		
D	Gathering data	4		
D	Making Booking	3		
D	Exception Handling	3		
D	Overall quality		4	
D	Total	20		
E	Off-line design and logbook writeup			2
E	Opening Window	2		
E	List item selection	3		
E	Details update	3		
E	Window closing	3		
E	Overall quality		2	
E	Total	15		

Part	Aspect	Marks		
		Correctness	Quality	Other
F	Off-line design and logbook writeup			1
F	Menu Items	2		
F	Load	2		
F	Store	2		
F	Exception Handling	2		
F	Overall quality		1	
F	Total	10		
	Maximum marks	100		

12o.9 Completing your work

As soon as you have completed, use `submit` to submit your work and then `labprint` to produce a listing ready for marking. The required files are named as follows.

```
BookFlightWindow.java
FlightBookingManagerGUI.java
ShowBookingsWindow.java
```

COMP16212 – Exercise 13o (0 Sessions – optional) Developing a Complete System: MP3 Database and Player

Alan Williams

13o.1 Aims

This laboratory represents the 'Grand Finale' for your programming and Java studies in COMP16121 and COMP16212. It enables you to consolidate your understanding of the various topics covered, by developing a complete system from scratch!

Of course, you will not be completely on your own: you will have seen and used virtually all of the techniques and constructs required to design and build the system, via the examples and associated tasks presented. In particular, the COMP16212 examples and tasks for files I/O, exception-handling, Collections and Advanced GUIs will be useful to you. We also give lots of hopefully useful hints on how to approach the design and implementation.

More than ever, you will need carefully to design your system away from the computer-face, *before* you actually start coding, making appropriate use of your log books.

13o.2 Learning outcomes

On successful completion of this exercise, a student will:

- have designed and implemented a larger Java example from scratch
- have utilised File I/O, Exceptions, Collections components, Advanced Graphics, and interfaces in a single application
- have used interfaces and other utilities provided with the example and from the Standard API
- have recalled and appropriately utilised previously covered topics and design examples
- have documented code and utilised existing javadoc documents
- be aware of HCI and Accessibility issues associated with Java-based graphical user interfaces

13o.3 Introduction

In this exercise, you will build a complete GUI-based MP3 song database and player, which allows a user to:

- load/save an existing MP3 song database
- filter, sort and view songs according to different criteria
- add, edit and delete songs
- play songs

The first half involves you designing the overall system and developing the model. In the second half you will develop a GUI for the model.

We have broken development into seven Stages A–G. The first part of the lab consists of Stages A–D and we estimate this will take you about half the time.

The second part consists of Stages E–G

We have included plenty of discussion on how to undertake the design and implementation. We have outlined the staged process of development and have provided some utilities to help you.

13o.3.1 Off-Line Preparation

Note that to start the exercise, and to complete Stage A (see below), you will need to undertake the problem analysis and design work. You should utilise off-line time to do this.

In particular, in off-line time you should undertake the following, recording as usual the results in your log book:

- Read through the whole of the exercise and ensure that you understand what is required
- List the various classes required and note how these are related — draw UML-style class diagrams.
- Ensure that you understand how the various given classes will be used in the system
- For each class, note the public methods required and describe their functionality
- For each class, consider the internal design, identifying any internal structures required. In particular, determine where Collections components could be used.
- Begin to develop pseudo-code for public methods and decide what internal helper methods would be useful
- Determine a set of appropriate test data to use throughout the exercise.

13o.4 Stages of development

The seven stages of the work are as follows:

A Overall Functionality and Design in logbook

B The `Song` class

C The `SongDatabase` class: The Model and Filtering

D The `SongDatabase` class: File Input/Output

E Exceptions

F The `SDBGUI` class: Basic GUI

G Extending the GUI: Viewing and Adding Songs

Important note: This is probably the largest exercise you have tackled so far. In particular, please bear in mind that there is more than can be done within the usual time for a single lab session. **This means we are really not expecting you to complete all of the exercise**.

We estimate that Stages A–C represent what, on average, you should be able to achieve in the single session. So we hope that:

- all of you will complete Stage A and get essentially to the end of Stage B,
- half of you will attempt Stage C,
- a few of you will get on to Stage D,
- and a handful will attempt any of Stages E/F/G.

The goal of the lab is to implement an application that allows the user to browse a collection of songs and to play the MP3 files for selected songs. Each song has some information associated with it, for example, the artist, title, genre, its URL for the MP3 file and the number of times it has been played. The user should be able to:

- load and save a song database
- display entries in the song database
- sort entries according to the various fields, e.g. show all tracks in order of track name;
- filter the display according to a combination of criteria. For example: show all tracks of genre “Easy Listening” which have the word “Moon” in the song title and the word “Frank” in the artist’s name.
- play selected tracks if there is an associated URL. A simple MP3 player, implemented in Java, is provided for you to do this.

There are several ways in which you could choose to tackle the exercise.

The application should consist of three basic components:

1. **The Model**

   The model will require classes representing (1) a song and (2) a collection of songs, with associated methods to manipulate these.

   Each song has information associated with it, which includes the following fields:

   - Title

- Artist
- Genre
- URL
- Times played

All of the above are strings, except for 'Times Played' which is an integer. Some of the fields could be empty. Optionally, this basic model can be extended to provide, for example, last date played, album and so on.

2. **File I/O**

   You will need to define an appropriate file format, along with functionality for reading/writing model objects.

   We have supplied some sample data in the file:

   ```
   /opt/info/courses/COMP16212/ex13o/MP3Player/sampleSongDatabase.mdb
   ```

   using a simple format described in Section 13o.5.2 below.

   This file includes some songs with viable URLs which you should be able to play[5], and some songs without viable URLs.

   You may wish to alter and/or extend the basic model, so you will need to think how to store that extended model, **but you should still be able to import the above data into your format** (or to develop a format that extends the format we provide).

   You will wish to add further data to the database in order to provide adequate test data for your system.

3. **GUI**

   You will need to develop a GUI that provides access to the functionality described above.

13o.5 The Laboratory Exercise

You should work in your `$HOME/COMP16212/ex13o` directory.

When compiling and running your code, you should set your *classpath* to include:

```
/opt/info/courses/COMP16212/ex13o/MP3Player/mp3player.jar
```

As before, there is no need to copy this jar-file across to your own directory — indeed it is not a good idea to do so.

As usual, to save you typing the same long commands repeatedly, it would be easiest for you to write simple shell scripts to compile and run your program.

You can find documentation for the various components that you will need to use at the following URL:

[5]The MP3 files involved are either from free download sites, or we have the owner's permission to use them

```
file:/opt/info/courses/COMP16212/ex13o/MP3Player/index.html
```

You should refer to this documentation for details of the various classes and methods to be used in your system. You should also refer to the Standard API documentation.

You should provide component-tests for the various stages, as well of course ensuring your complete system functions correctly.

13o.5.1 Stage A: Overall Functionality and Design

As described above, on completion, your MP3 database and player system should have the following functionality:

- load a plain-text song database file
- save a plain-text song database file
- view the list of songs in the database
- display the list of songs, sorted according to title, artist, genre or number of times played
- select a song from the list
- view the details of the selected song
- play the selected song
- filter the list of songs viewed, matching on a combination of genre, artist and title
- delete a selected song
- add a new song
- edit an existing song

As with the Diary example and Flight-booking exercise, you should aim to have a clean separation between the model (the collection of songs and associated functionality) and the GUI.

For successful completion of Stage A, you should produce the top-level design of the overall system within your logbook, including pseudo-code and test data. Clearly, this should be done *before* detailed implementation work.

13o.5.2 Stage B: The `Song` class

You should produce a class called `Song` which provides the basic facilities for creating and processing song records.

In order to get you started, we have provided a template class `SongTemplate` which we recommend you copy, to file `Song.java` and then modify in order to do this. In the following description, we assume you do this.

So you should begin by copying the template song file:

```
/opt/info/courses/COMP16212/ex13o/MP3Player/SongTemplate.java
```

to `$HOME/COMP16212/ex13o/Song.java` and studying the code, and documentation available from URL:

```
file:/opt/info/courses/COMP16212/ex13o/MP3Player/index.html
```

In particular, this class provides various class constants for determining the fields used for matching and selection. It should also provide accessor methods for the fields and implement the `Comparable<Song>` interface (stubs are given).

You should work out what `Song` constructor methods you will require: you will need to be able to build a song object from a `String` representation of a song (i.e. as read in from a file), as well as more directly from the different fields (i.e. as supplied by the GUI).

The file format of the test song database is a plain-text file:

```
DBTITLE=[DATABASE TITLE]
[SONG_1]
[SONG_2]
...
[SONG_n]
```

where '`[DATABASE TITLE]`' is the database title and should be displayed in your GUI, and each '`SONG_i`' has the following format:

```
TITLE<tab>ARTIST<tab>GENRE<tab>URL<tab>TIMESPLAYED
```

Note that '`<tab>`' is the tab character, '`TITLE`', '`ARTIST`', '`GENRE`', '`URL`' are strings giving the various song fields, and '`TIMESPLAYED`' is an integer. See the example database at:

```
/opt/info/courses/COMP16212/ex13o/MP3Player/sampleSongDatabase.mdb
```

In order to parse a song represented as a string, you could use the various methods from the `String` class, such as '`split("\\t")`', where the argument represents the *regular expression* describing the strings separating the components. This returns an array of strings, which should be the fields within a song.

In terms of methods, you will need at least to (1) order songs, (2) match songs (3) display songs in various ways, i.e. as strings.

You will need to provide ordering methods for your `Song` objects, capable of sorting them according to the required 'sort-mode' (e.g. ordering first by artist, or by genre, or by another field, and so on). In order to enable collections to do the sorting for you, your class will need implement `Comparable<Song>`. This will need to be a total ordering. One possible way of implementing this is that you could create a string composed of all the fields in the song and then simply use string comparison within your definition of your song comparison. Your `compareTo()` method will take a `Song` parameter.

To start with, you could ignore changing the sort-mode and just provide an ordering based on all the song's fields. You could then later extend this to take into account the sort mode — the following pseudo-code could form part of your `Song` design:

```
new class variable: int currentSortMode
   this will be one of the sort-mode constants
   as defined in Song.java
```

```
public method setCurrentSortMode(sortMode):
   sortMode will be one of the sort-mode constants

   assign sortMode to currentSortMode

Within the compareTo() method:
   switch statement depends on currentSortMode:
      this will determine the order in which fields are compared
      e.g. create strings from the fields of the two songs in the required order
      default order?: artist+title+genre+url+numberOfTimesPlayed

   calculate and return the result of comparing the two strings
      (can use string compareTo() already defined in String)
```

Why do we need to have a separate variable to hold the sort-mode?

What methods will you require to perform matching? Will matching be exact or partial, via sub-strings (or both?). As with comparison, the result of matching will be mode-dependent. The match method could take two arguments: (1) the string to use in the match (2) the match mode. So the possible pseudo-code could be:

```
(*assume we only need to match on string fields!* i.e. not numberOfTimesPlayed)

boolean method matches(String stringToMatch, int matchMode)
  matchMode will be one of the mode constants
  as defined in Song.java

  switch statement on matchMode:
    in each case:
       retrieve field from this song required by matchMode

       if an exact match is required, according to matchMode:
         field must be equal to stringToMatch
       if a partial match is required, according to matchMode:
         use indexOf() method from String and check for
             index >= 0.
```

The `indexOf(stringToMatch)` method will return the index of the start of `stringToMatch` within the string to which it is applied (see the Standard API documentation for `String` for details).

Do you require any mutator methods? Do you wish to extend this model? (for example, editing might be achieved within the database simply by deleting the old version and adding the new version of a song)

Now determine any additional methods you require.

In particular, you will need to provide additional methods to preserve the required consistency with `compareTo()`.

Finally you will need to display your songs in various ways as strings. We suggest you use the default `toString()` method to show a song in a suitable format for displaying in a list of songs (e.g. this may not include all the information, such as the URL). This is because classes such as `JList` utilise by default the `toString()` method to display elements. You would then need to provide another string-producing method for use when saving a song to a file (this of course should produce the song in the format expected when you load a file).

You should design test data and expected results and later include these in a `main` method. Clearly, you need to carefully test your constructor, comparison and matching methods.

See the documentation for the original `SongTemplate` class for specifications of the various suggested methods involved:

```
file:/opt/info/courses/COMP16212/ex13o/MP3Player/SongTemplate.html
```

13o.5.3 Stage C: The `SongDatabase` class: The Model and Filtering

In this part you will begin development of the `SongDatabase` class. You should design and develop methods for representing and manipulating the song database model, i.e. the collection of songs within the database. You should aim for your model to provide a clean interface for your GUI, so that the GUI does not need to be aware of internal structures.

You should determine the best `Collection` component to use to represent internally the complete set of songs. For example, under what circumstances does this need to be ordered? Should it be a set or a map? Should it allow repeats?

What methods are associated with this? To start with, you will need to add and delete songs, provided by your GUI in some way (recall the Film Database example from COMP16121).

Now, how will you provide filtering? You will need to store an internal *copy* of the complete set of songs, which is taken each time the filter is reset. How will you represent this copy. Does it need to be ordered? Now, each time a filter is required, it will be applied to this copy to produce a sub-set of songs to be stored ready for the next filter. For example, you may wish to filter on genre, then filter the result on artists, then filter that result on title.

The core of this functionality would be a filter method that takes a ‘match mode’ and filter string as argument (assuming the filter is always going to be a string). Here is some possible outline pseudo-code for this part of the design of `SongDatabase`:

```
new instance variable: filteredSongs
   this will be a Set (should it be sorted or not?)

public method: resetFilter()
   create a copy of the collection of songs and assign to filterSongs
      (use Collection constructor which takes another Collection as arg)

public method: filterSongs(matchMode, filterString):
   check if filterString is non-empty (only proceed if it is)
   create a new empty set of songs: newSet
   iterate through all current filterSongs:
```

```
        if (next song matches on filterString according to matchMode)*
           add to newSet
     assign newSet to filteredSongs

  *method provided by Song class

  Also: public method needed to return filterSongs as an array
          (sorted)
```

The above relies on the song matching method provided in the `Song` class.

The next time the `filterSongs` method is called, it will start with the last set of filtered songs.

The GUI access to the filtered songs is likely to be as an array (e.g. you may well be adding them to a `JList`), so you will need to convert your filtered sub-set of songs into an array. Does this need to be ordered and how will this be achieved?

Your filter will need to be reset when songs are added or deleted, as well as when a new filter is requested by the GUI.

Since you need to be able to select on different fields, you will probably need to give the GUI access to field lists for the currently filtered songs (e.g. the list of genres or artists contained in the filtered songs). Again these are likely to be provided as arrays. You will therefore need public methods to provide these.

Finally, you will need to implement the play mechanism for a song. This is reasonably straightforward: the method will be provided with an instance of `MP3Player` and simply provide its `play` method with the URL for the song you wish to play. Note that the GUI controller itself should create the `MP3Player` instance (or at least have access to it), since it may need to be able to stop the song, or will need to listen for end-of-play. See the documentation for `MP3Player` for more details:

```
file:/opt/info/courses/COMP16212/ex13o/MP3Player/MP3Player.html
```

You should design test data and expected results and later include these in a `main` method.

13o.5.3.1 Filtering Example

Here is a small example of filtering, using a contrived example song database:

1. Suppose we start with the following song database. We therefore initialise the filter set to contain these:

Title	Artist	Genre
Mars Landing	Howard	Classical
Mars	Howard	Classical
Earth	Howard	Classical
Mars Landing	Alan	Classical
Earth	John	Classical
Venus	Howard	Folk
Mars	John	Folk
Venus	John	Rock

2. We now apply a partially matching filter to the Title field using the string ‘Mars’, giving:

Title	Artist	Genre
Mars Landing	Howard	Classical
Mars	Howard	Classical
Mars Landing	Alan	Classical
Mars	John	Folk

3. To the above result, we now apply the second filter to the Genre field using the string ‘Classical’, giving:

Title	Artist	Genre
Mars Landing	Howard	Classical
Mars	Howard	Classical
Mars Landing	Alan	Classical

4. To the above result, we now apply the third filter to the Artist field using the string ‘Howard’, giving:

Title	Artist	Genre
Mars Landing	Howard	Classical
Mars	Howard	Classical

The resulting filtered list of two songs can then be retrieved from the model and displayed by the GUI. Each time a filter string is changed in the GUI, the above process can be applied.

Also, convince yourself that it does not in fact matter which order the filters were applied — the same two songs will remain.

13o.5.4 Stage D: The `SongDatabase` class: File Input/Output

We suggest you develop the file output functionality first of all, implementing the `save()` method:

```
public void save(PrintWriter writer)
```

The GUI will later need to obtain and provide the `writer`.

You then need to be able to load a saved data file:

```
public void load(BufferedReader reader)
```

The GUI will later need to obtain and provide the `reader`.

In particular you will also need to be able to load the test data file at:

```
/opt/info/courses/COMP16212/ex13o/MP3Player/sampleSongDatabase.mdb
```

You should design test data and expected results and later include these in a `main` method.

13o.5.5 Stage E: Exceptions

In your design, you will have identified where different kinds of exception may arise. You should now develop the necessary exceptions classes and modify your code so that these are generated and handled appropriately.

For example what exceptions may arise when loading a file? Where should these exceptions be handled?

13o.5.6 Stage F: The `SDBGUI` class: Basic GUI

You should carefully plan the design of your GUI so that it can provide the required functionality in an easy-to-use manner. In particular, you should utilise guidelines from good Human-Computer Interaction design when doing this.

For this stage you should aim to:

- set the GUI title to be the database title
- display the current (filtered) list of songs, according to the currently selected ordering
- define filters on genre, artist and title
- specify different display orderings
- load and save a database
- select a song from the displayed list
- delete a selected song
- play a selected song

Which GUI components are going to be most appropriate? What layout managers would be most appropriate? How are you going to provide the listeners to invoke the required functionality?

Here are some suggestions for approaches:

- use a `JList` with scroll-bars to view the list of songs (e.g. see the `Diary` or flight booker examples). The `toString()` method of a song will, by default, determine its appearance in the list (an alternative but more involved approach would be to display the list elements in a format different from their natural string appearance (see on-line documentation for this)
- use groups of radio buttons to select the song order (again see the `Diary` example)

- There are various ways to specify and invoke filters — somehow you need to provide a GUI component that allows the user to specify a string. For example, use separate `JTextFields` for each criteria (genre/artist/title), which can take a string giving the required match. Then have a 'filter' button to separately invoke the filter. Or perhaps you could use a `DocumentListener` on the `JTextField` to directly listen for any changes to the filter string? Should the matching be exact, sub-string, case-sensitive?

  You will need to apply a sequence of the different filters, for each field. For example, when any filter changes, the following simple approach could be taken:

  ```
  When a filter changes:
     reset database filter (so that it holds all songs)
     retrieve and apply Title filter string
     retrieve and apply Genre filter string
     retrieve and apply Artist filter string
     retrieve resulting array of filtered songs from filter
        collection, respecting current sort mode, and display
  ```

- use `JMenu` for loading and saving, using a `JFileChooser` object so that, for example, visited files and directories are remembered.

HCI and Accessibility Issues: You should write a short report in your logbook analysing your GUI with respect to HCI issues, in particular Accessibility. Here, you should point out any problems with your system, HCI-wise. However you do not need to fix these in your implementation. For example, you may determine that the font on your GUI is too small, but you do not then need to provide a font size selector. There may also be HCI-related activities that you ideally would like to have carried out, so you should make a note of these. Accessibility-wise, you should aim to analyse your GUI using an appropriate Java accessibility assessment tool. You should aim to fix problems highlighted (or note problems that are raised).

13o.5.7 Stage G: Extending the GUI: Viewing and Adding Songs

You should provide separate windows for viewing and adding songs. The viewer should also allow you to start and stop playing the song.

13o.6 Optional Parts

In no particular order, here are some suggestions for extending your GUI:

- editing song
- provide help
- using `JList`-based filters rather than `JTextField`, with exact matching, so that you can display to the user the available values. You could provide lists of the values for the different fields in the current filtered song list, and then provide a mechanism for selecting the filter value from the list. For example, you could use a key event listener on the field

list, listening for the ‘Enter’ key to be pressed for example, then retrieve any currently selected field value to act as the filter.

One problem is that, by default, once a selection has been made in a `JList` you cannot undo it. However, see

```
file:/opt/info/courses/COMP16212/ex13o/MP3Player/ToggleList.html
```

for a description of the `ToggleList` class which extends `JList`, allowing you to ‘toggle’ the selected element. (`ToggleList` is included in `mp3player.jar`)

Your underlying model would also need to be able to provide the arrays of field values extracted from the current filtered song list.

- parse checks and exception-handling
- play listeners: dis/enabling of play/stop buttons
- multiple viewers/players
- counting songs displayed
- checking for changes to the database before load/quit
- checking for and disposing viewer/editor windows on load
- clearing filters on load
- use of the `Observer` class (seen in the Diary example)
- setting and change the database title
- load-merge, to merge two databases
- use of `JSplitPane`
- case-sensitive matching
- add extra information to the database, such as Albums and ratings fields.

13o.7 Assessment criteria

We will mark the exercise at the end

The exercise is marked out of 125 marks, and is worth one **optional session** of the assessment for the COMP16212 laboratory.

The following is a marks allocation for this exercise:

Part	**Aspect**	**Marks**	Totals
A	Off-line Design and write up in logbook	10	
A	Design write-up Total		10

Part	Aspect	Marks	Totals
B	comparison methods of `Song`	10	
B	filtering methods of `Song`	10	
B	constructor methods of `Song`	10	
B	sort and match mode methods	5	
B	to-string methods of `Song`	5	
B	Overall test and design	5	
B	Song Total		45
C	`SongDatabase`: basic model	5	
C	`SongDatabase`: filter	10	
C	Overall quality and test	5	
C	SongDatabase Total		20
D	`SongDatabase`: file load	5	
D	`SongDatabase`: file save	5	
D	File I/O Total		10
E	Handling Exceptions	5	
E	Exceptions Total		5
F	GUI(1): set GUI title	2	
F	GUI(1): display list	4	
F	GUI(1): select film and delete	4	
F	GUI(1): different list ordering	4	
F	GUI(1): first filter	4	
F	GUI(1): combined filtering	2	
F	GUI(1): HCI and Accessibility	5	
F	GUI(1) Total		25
G	GUI(2) View Film and play	5	
G	GUI(2) Add Film	5	
G	GUI(2) Total		10
	Total Maximum marks		125

13o.8 Completing your work

As soon as you have completed, use `submit` to submit your work and then `labprint` to produce a listing ready for marking. The required files are named as follows.

```
Song.java
SongDatabase.java
SDBGUI.java
```

You can also include the optional file `othercode.txt` which should hold source for any additional classes you may have defined while tackling this exercise. You can produce this file

using the '`cat`' command. For example if you have additional Java files `MyClassA.java`, `MyClassB.java`, `MyClassC.java` then you can create the text file with the following command:

```
cat MyClassA.java MyClassB.java MyClassC.java > othercode.txt
```