

COMP1500/COMP7901

Assignments

Second Semester 2004

Assignment 1

Before attempting this assignment you will need to complete your study of *Java Genesis* up to and including Chapter 4.

PROBLEM: expanding circles



Open the folder *genesis/projects/Assignments/Expanding Circles* and run the class *Demo*. A window opens containing a circle with radius initially 5 pixels and with the x-coordinate of its centre randomly chosen in the range 50 to 342 pixels inclusive and the y-coordinate of its centre randomly chosen in the range 50 to 218 pixels inclusive. The colour of the circle is randomly chosen from the colours red, blue, green, magenta and yellow. The circle immediately starts to grow at the rate of 1 pixel every 20 milliseconds until the edge of the circle first touches one of the four borders of the display area. At this point the circle stops expanding and stays unchanged for 500 milliseconds, after which the whole process is repeated. That is, the radius of the circle is set to 5 pixels, the x-coordinate of its centre is randomly chosen in the range 50 to 342 pixels, the y-coordinate of its centre is randomly chosen in the range 50 to 218 pixels, and its colour is randomly chosen from those four colours in the collection red, blue, green, magenta and yellow different from the current colour. For example, if at some stage the circle is green, say, then at the next stage the colour chosen for the circle must be either red, blue, magenta or yellow chosen randomly, but not green. The circle then expands as before until it first touches one of the four borders of the display area, and so on, repeating the process continuously until the program is terminated by the user.

The assignment task

Your task is to create a class *ExpandingCircles* which when compiled and run has the same behaviour as the demonstration. You may find Problems 3, 13 and 16 of Chapter 4 of *Java Genesis* and Lab Assessment 3 helpful with this assignment. In particular, the window is opened by executing the statement

```
CircleFigure.create( );
```

and the circle's colour, size and position can be set by sending appropriate messages to the class `CircleFigure`. Consult Section 2.4 of *Java Genesis* to recall the various messages that can be sent to this class. For this assignment you may assume that the display area is 392 pixels wide and 268 pixels deep.

Grading the assignment

The following marking scheme will be used when grading your assignment. The assignment is worth 10%.

- 1 Mark** if the window opens correctly when your class `ExpandingCircles` is compiled and run.
- 2 Marks** if initially the centre of the circle is set so that its x-coordinate is randomly chosen in the range 50 to 342 pixels inclusive, its y-coordinate is randomly chosen in the range 50 to 218 pixels inclusive and its radius is 5 pixels.
- 2 Marks** if its colour is randomly chosen from the colours red, blue, green, magenta and yellow.
- 2 Marks** if the circle grows at the rate of 1 pixel every 20 milliseconds until it first touches one of the four borders of the display area.
- 1 Mark** if the circle then stays unchanged for 500 milliseconds.
- 1 Mark** if the whole process continually repeats until the application is terminated by the user, with at each stage the x-coordinate of the circle's centre randomly chosen in the range 50 to 342 pixels, the y-coordinate of its centre randomly chosen in the range 50 to 218 pixels and its radius 5 pixels.
- 1 Mark** if the circle's new colour is randomly chosen from those four colours in the collection red, blue, green, magenta and yellow different from the current colour.

Important information

When you have completed this problem, add your name and student number as a comment at the top of the file *ExpandingCircles.java* and submit this file using the on-line submission system which you can access from the course web-page. Your submitted file must have this name. **Note:** make sure you record your Submission ID: this is your proof of submission.



You must submit your assignment before the deadline of

4pm Friday August 20th.

Late assignments will not be accepted.

Assignment 2

Before attempting this assignment you will need to complete your study of *Java Genesis* up to and including Chapter 7.

PROBLEM: walking the plank



Pirates walk the following plank:

0	1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	---	----

A pirate starts at position 0 and at each stage steps with equal probability either one position to the left or one position to the right, except that

- when at position 0 the pirate can step only to position 1;
- when at position 10 the pirate steps neither to the right or left but stays there (reaching position 10 can be thought of as the pirate falling off the plank into the water).

The assignment task

Open the folder *genesis\projects\Assignments\Walking The Plank*. Your task is to create in this folder classes `Pirate`, `PlankStatistics` and `PercentSaved`.

Coding the class `Pirate`

Objects of the class `Pirate` represent pirates. The state and behaviour of a pirate will be captured by the instance variables and methods of this class. In order to ensure that pirates have the required behaviour, the class `Pirate` will need to satisfy the following specification.

- The class `Pirate` will need to have two instance variables:
 - `posn`
an integer denoting the current position of the pirate;
 - `numOfVisitsToPosn`
an array of integers of length 11 to record the number of times each position has been visited by the pirate. That is, for any i between 0 and 10, `numOfVisitsToPosn[i]` is set to k , say, if the pirate has visited position i k times.
- The class `Pirate` will need to have a constructor method that sets appropriate initial values for the variables `posn` and `numOfVisitsToPosn`. To be specific, a pirate starts walking from position 0, and at this stage has visited only position 0. Hence `posn` is initially 0 while `numOfVisitsToPosn[0]` is initially 1 but all other entries in the array are initially 0.

- The class `Pirate` will need to have four instance methods:

`step()`

returns no value but results in the current position of the pirate changing to one of the neighbouring positions as described earlier, i.e. the pirate steps with equal probability either one position to the right or one position to the left except that when at position 0 the pirate can step only to position 1 and when at position 10 the pirate stays there and steps no more (as the pirate either drowns or is eaten by the sharks thrashing about in the water below). When a pirate steps to position i , say, `numOfVisitsToPosn[i]` is increased by 1;

`toString()`

returns a string giving information about the current state of the pirate (see below for details of what information needs to be captured by this method).

`getPosn()` and `getNumOfVisitsToPosn()`

getter methods that return the values of the variables `Posn` and `numOfVisitsToPosn` respectively.

So that you know what your `Pirate` class is meant to do, and in particular what information needs to be captured by the `toString` method, we have placed a class `WalkThePlank` in the folder `genesis\projects\Assignments\Walking The Plank`. Here is the code for this class:

```
import genesis.*;

public class WalkThePlank {

    public static void main (String [ ] args) {
        Pirate pete = new Pirate( );
        for (int i=1; i<=60; i++) pete.step( );
        Transcript.println(pete);
    }
}
```

This class creates an object `pete` of the class `Pirate` and gets `pete` to take 60 steps along the plank.

After you have completed coding your class `Pirate`, compile the classes `Pirate` and `WalkThePlank`, select the class `WalkThePlank` as the Main Class and run the application. If you have coded the class `Pirate` correctly, output similar to the following should appear in the Transcript window (your output may well be different depending on the random steps taken by pirate `pete`). We ran our `WalkThePlank` class twice; the first execution resulted in the output

```
pirate's current position: 10
pirate has fallen off the plank
number of visits to position 0: 1
number of visits to position 1: 2
number of visits to position 2: 4
number of visits to position 3: 6
number of visits to position 4: 11
number of visits to position 5: 13
number of visits to position 6: 6
number of visits to position 7: 1
number of visits to position 8: 1
number of visits to position 9: 1
number of visits to position 10: 1
```

while the second execution resulted in the output

```
pirate's current position: 8
pirate is still walking
number of visits to position 0: 1
number of visits to position 1: 1
number of visits to position 2: 8
number of visits to position 3: 13
number of visits to position 4: 12
number of visits to position 5: 9
number of visits to position 6: 4
number of visits to position 7: 4
number of visits to position 8: 6
number of visits to position 9: 3
number of visits to position 10: 0
```

Coding the class PlankStatistics

The class `PlankStatistics` contains just a `main` method. When this class is compiled and run, pirate `pete` walks the plank 10,000 times, in each case walking until position 10 is reached. The average number of visits to each position, rounded to the nearest integer, is computed and output. That is, each of the 10,000 plank walks continues until `pete` reaches position 10. For each of these 10,000 walks and for each `i` between 0 and 10, the number of visits to position `i` is extracted, these 10,000 values are then added and divided by 10,000 to give a floating point number average. This average is then rounded to the nearest integer and output to the Transcript window. For example, when we compiled and ran our `PlankStatistics` class we got the following output in the Transcript window:

```
average number of visits to position 0: 10
average number of visits to position 1: 18
average number of visits to position 2: 16
average number of visits to position 3: 14
average number of visits to position 4: 12
average number of visits to position 5: 10
average number of visits to position 6: 8
average number of visits to position 7: 6
average number of visits to position 8: 4
average number of visits to position 9: 2
average number of visits to position 10: 1
```

The output from your class should have the same form as this, although the actual numbers calculated may be slightly different.

(Note: probability theory suggests that the average number of visits by a pirate to position i where $1 \leq i \leq 9$ is $2(10 - i)$, while the average number of visits to position 1 is 10 and to position 10 is 1.)

Coding the class PercentSaved

The naval captain who has captured the pirates has a forgiving nature. From the previous statistics he decides that as on average a pirate will visit position 0 ten times, he will pardon and free any pirate that visits position 0 for the tenth time. The captain argues that as ten is the average number of visits to position 0, half the pirates will visit position 0 ten times before reaching position 10, and hence half the pirates will be pardoned rather than drowned (or eaten by the sharks). Is the captain correct in his calculation? Write a program `PercentSaved` containing just a `main` method which when compiled and run gets pirate pete to walk the plank 10,000 times, in each case walking until either position 10 is reached (and pete is drowned or eaten by sharks) or position 0 is visited for the tenth time (and pete is saved from a horrible death). In each case a record is kept of whether or not pete is saved, and after the 10,000 trials are completed, the percentage of times pete is saved is printed in the Transcript window. For example, when we compiled and ran our `PercentSaved` class we got the following output in the Transcript window:

```
saved: 38.24%
```

Grading the assignment

The following marking scheme will be used when grading your assignment. The assignment is worth 10%.

5 Marks for coding the `Pirate` class, allocated as follows:

- 1 Mark** for correctly declaring the instance variables `posn` and `numOfVisitsToPosn`;
- 1 Mark** for correctly coding the constructor method;
- 1 Mark** for correctly coding the `step` method;
- 1 Mark** for correctly coding the `toString` method;
- 1 Mark** for correctly coding the getter methods `getPosn` and `getNumOfVisitsToPosn`.

3 Marks for coding the `PlankStatistics` class, allocated as follows:

- 2 Marks** for calculating the averages correctly;
- 1 Mark** for producing output to the Transcript window of the correct format as illustrated in the examples above.

2 Marks for coding the `PercentSaved` class, allocated as follows:

- 1 Mark** for calculating the percentage correctly;
- 1 Mark** for producing output to the Transcript window similar in structure to that illustrated in the example above.

Important information

When you have completed this problem, add your name and student number as a comment at the top of your three modified files *Pirate.java*, *PlankStatistics.java* and *PercentSaved.java* and submit these files using the on-line submission system which you can access from the course web-page. Your submitted files must have these names. **Note:** make sure you record your Submission ID: this is your proof of submission .



You must submit your assignment before the deadline of

4pm Friday September 10th.

Late assignments will not be accepted.

Assignment 3

Before attempting this assignment you will need to complete your study of *Java Genesis* up to and including Chapter 10.



PROBLEM: colour all green

To explore the colour-all-green puzzle, run the class `Demo` which is in the folder `genesis\projects\Assignments\Colour All Green`. A window opens displaying a 5 by 5 grid of blank squares. Above the grid appears the message ‘click on any blank square to turn it green’. As this message suggests, if the mouse is positioned on any blank square and the mouse button pressed, that square turns green. Furthermore, each immediate horizontal or vertical coloured neighbouring square (i.e. any square that is no longer blank and is immediately above or below, or immediately to the left or right of the chosen square) has its colour toggled between green and red (i.e. if it is green it turns red; if it is red it turns green). The blank neighbouring squares and all other squares remain unchanged. Notice that if the mouse is clicked on an already coloured square nothing happens.

The aim of the puzzle is to end up with all the squares coloured green after the mouse is clicked upon the last of the blank squares. If indeed at this stage all the squares are coloured green, the message ‘PUZZLE SOLVED’ is displayed. However, if at this stage not all the squares are green (i.e. at least one is red), the message ‘puzzle NOT solved’ is displayed.

The window also contains a ‘reset’ action button. If at any time this button is pressed, each square once again becomes blank, i.e. the initial configuration is restored and the puzzle is ready to be tried again. The original message ‘click on any blank square to turn it green’ is displayed.

In addition, the window contains two text fields each initially displaying the integer 5 and labeled respectively ‘number wide’ and ‘number high’. If an integer n is inserted into the ‘number wide’ text field and an integer m into the ‘number high’ text field, when the ‘reset’ button pressed

- if n and m are each in the range 1 to 9 inclusive a new puzzle is created consisting of an n wide and m high grid of blank squares;
- if $n > 9$ (or $m > 9$) the number in the respective text field is replaced by 9 and a new puzzle is created as in the previous case;
- if $n < 1$ (or $m < 1$) the number in the respective text field is replaced by 1 and a new puzzle is created as in the first case.

Attempt this puzzle with various values of n and m until you understand the way the interface works. Notice that the grid of squares in a puzzle is positioned in the centre of the window.

The assignment task

In the folder *genesis\projects\Assignments\Colour All Green* you will find the classes `Main`, `PuzzleFrame` and `SquarePanel`. If you run the `Main` class a window opens displaying a 5 by 5 grid of blank squares. Above the grid appears the message 'click on any blank square to turn it green'. However, if the mouse is clicked on any square nothing happens. Below the grid of squares is the action button 'reset'. If this action button is pressed again nothing happens.

Your task is to modify the code for the two classes `PuzzleFrame` and `SquarePanel` so that when these classes are compiled and the `Main` class run, your application has the same appearance and behaviour as the demonstration.

Grading the assignment

The following marking scheme will be used when grading your assignment. The assignment is worth 10%.

- 1 Mark** if the two classes `PuzzleFrame` and `SquarePanel` compile successfully and when the application is run a window opens displaying a 5 by 5 grid of blank squares, with the action button 'reset' underneath and the message 'click on any blank square to turn it green' displayed above, as in the demonstration. Note that if these two classes do not compile then no marks will be awarded for the assignment.
- 1 Mark** if whenever the mouse is positioned over any blank square and the mouse button pressed, that square turns green.
- 1 Mark** if whenever the mouse is positioned over a square already coloured green or red and the mouse button pressed, nothing happens.
- 1 Mark** if whenever a blank square is first coloured green, each immediate horizontal or vertical neighbouring square which is already coloured has its colour toggled between green and red, while all other squares are unchanged.
- 1 Mark** if the message 'PUZZLE SOLVED' is displayed whenever all the squares end up green after the mouse is clicked upon the last of the blank squares.
- 1 Mark** if the message 'puzzle NOT solved' is displayed whenever at least one of the squares is red after the mouse is clicked upon the last of the blank squares.
- 1 Mark** if whenever the action button 'reset' is pressed, each square becomes blank and the message 'click on any blank square to turn it green' is displayed, and the puzzle is ready to be tried again.
- 1 Mark** if there are two text fields labeled 'number wide' and 'number high' which each displays 5 initially. If an integer n , say, is inserted into the 'number wide' field and an integer m , say, is inserted into the 'number high'

field and the 'reset' button pressed, a grid of squares n wide and m high is displayed. If, however, $n > 9$ (or $m > 9$) the number in the corresponding text field becomes 9 and the grid of squares has width (respectively height) 9. Similarly, if $n < 1$ (or $m < 1$) the number in the corresponding text field becomes 1 and the grid of squares has width (respectively height) 1.

2 Marks if the puzzle works not just for the 5 by 5 grid but for any grid with width and height between 1 and 9 (i.e. clicking with the mouse on a blank square turns it green and toggles its coloured neighbours, and when the last blank square is clicked the appropriate message is displayed; furthermore, pressing the 'reset' button restores the same sized grid to its initial state, as in the demonstration).

Important information

When you have completed this problem, add your name and student number as a comment at the top of your two modified files *PuzzleFrame.java* and *SquarePanel.java* and submit these files using the on-line submission system which you can access from the course web-page. Your submitted files must have these names. **Note:** make sure you record your Submission ID: this is your proof of submission.



You must submit your assignment before the deadline of

4pm Friday October 15th.

Late assignments will not be accepted.

Postscript

As you will no doubt have noticed, for some grid sizes the puzzle can be solved, and for other sizes it cannot, i.e. it is not always possible to have all the squares end up green after the last square has been coloured. However, there is a simple condition on the grid size that enables us to predict whether or not the puzzle can be solved. Although not part of this assignment, for your own satisfaction see if you can discover this condition for solvability of the puzzle, and in the cases when the puzzle can be solved, describe a general strategy for ensuring that all the squares end up green.