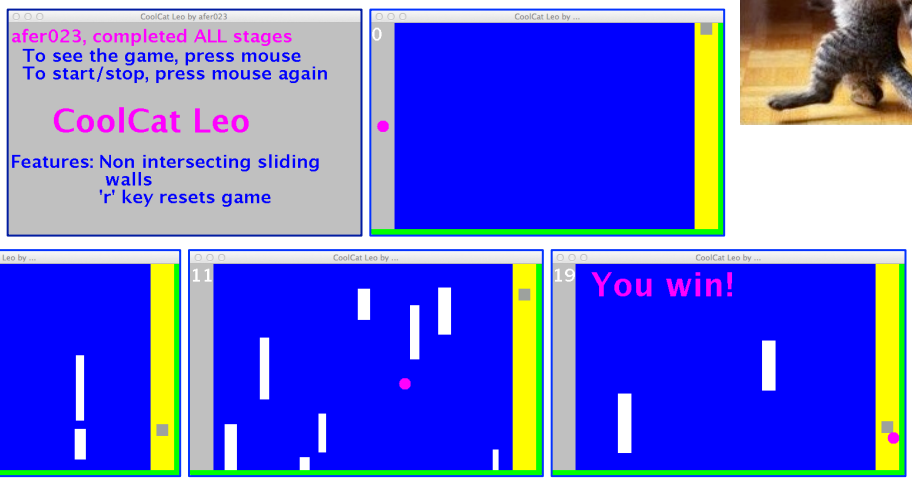
COMPSCI 101 – Semester 2, 2013

Assignment Three – CoolCat Leo



Due: 4:30pm, Friday 25th October
Worth: 6%

Introduction

In this assignment, you will develop a game in which the user (the `CoolCat` object) aims to move across the game area to reach the home square without running into any of the vertically sliding walls. The screenshots above show just one possible solution to this assignment.

Learning goals

This assignment is designed to help you learn a number of concepts and practise a number of skills. In particular:

- to understand event handling
- to deal with animation and graphics
- to organise code into appropriate classes and methods

Important things to note

- In this assignment you MUST use `Rectangle` objects to store the current position of the `CoolCat` object, the current position of each `SlidingWall` object and the current position of the `HomeSquare` object.

- Skeleton files for all the classes used in this assignment have been provided. You may choose to add other Java classes of your own.
- You have been given the `A3Constants` class, which contains many public constants which can be used when developing the 'CoolCat Leo' game, e.g.,

  `LEO_LEFT_AREA`: a `Rectangle` object defining the left hand part of the game area.
  `LEO_START_AREA`: a `Rectangle` object defining the initial position of the `CoolCat` object.
  `ALL_WALLS_AREA`: a `Rectangle` object defining the middle part of the game area.
  `HOME_AREA`: a `Rectangle` object defining the right hand part of the game area.
  `HOME_START_AREA`: a `Rectangle` object defining the initial position of the `HomeSquare` object.
  `UP, DOWN, LEFT, RIGHT`: constants used to set and test the direction of the game objects.

  There are many other constants, not mentioned above, which you can use. Examine the `A3Constants` class to see which constants have been defined. To refer to the constants in the `A3Constants` class, prefix the name of the constant with the classname, e.g., `A3Constants.UP, A3Constants.LEO_START_AREA`. You may change or add any constants you need to the `A3Constants.java` file.

For this assignment, there is a minimum amount that you are required to do. You are encouraged to extend your code in any way you please but YOU MUST NOT USE IMAGES OR SOUNDS. Please read the assignment document carefully to make sure that you complete all the minimum requirements.

It is strongly recommended that you develop your program in stages as described in this document.

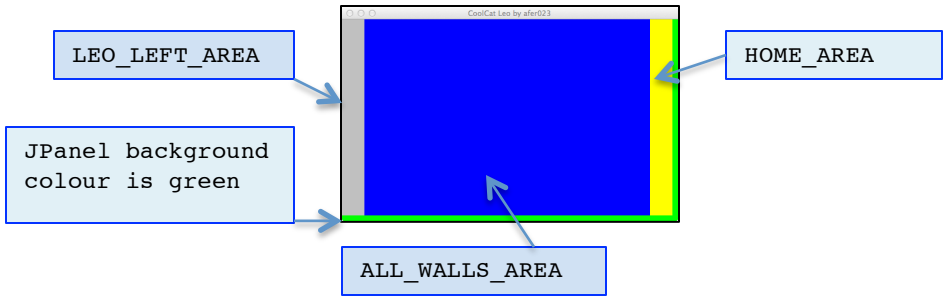
You should only submit the final Java classes of your assignment. Download the Assignment 3 folder.

Stage 1 – Your UPI

Your UPI should be displayed inside the title bar of the window. To do this, open the `A3.java` file and replace the "..." after "`CoolCat Leo by ...`" with your UPI:

```
JFrame gui = new A3JFrame("CoolCat Leo by ...", ... );
```

You may name your game (and player) anything you like.



Stage 2 – Display the game area

The game has three areas:

On the left there is a thin strip (the `LEO_LEFT_AREA`) which is the starting area for the player object (the `CoolCat`).

In the middle there is the main area (the `ALL_WALLS_AREA`), through which the player has to pass (avoiding the sliding walls).

To the right there is a thin strip (the `HOME_AREA`), which is the area in which the home square moves up and down.

In the `A3JPanel` class fill the three parts of the game areas with colours. You MUST use the `Rectangle` constants, HOME_AREA, ALL_WALLS_AREA, LEO_LEFT_AREA in your code.

In your `paintComponent()` method be sure to make a call to a helper method:

```
private void drawGameArea(Graphics g) { ... }
```

Please note that for this game you may use any colours of your choice.

Stage 3 – Define the CoolCat class

The user controls a `CoolCat` object which moves in all four directions (up, down, left, right) as it tries to move across the middle of the game window in order to get to the home square. For this stage of the assignment you need to define the `CoolCat` class. The skeleton of the `CoolCat` class is shown below:

```
public class CoolCat {
   private Rectangle area;
   private int speed;
   private int direction;

   public CoolCat() { /* constructor */  }

   public Rectangle getArea() { ... }
   public void setDirection(int direction) { ... }
   public boolean hasReachedHome(Rectangle homeArea) { ... }
   public void move() { ... }
   public void draw(Graphics g) { ... }
}
```

Things to note about the CoolCat class

The `CoolCat` class has 3 instance variables.

The `area` instance variable represents the current position, width and height of the `CoolCat` object. Initially the `CoolCat` object is positioned in the area given by the constant, `LEO_START_AREA`.

The `direction` instance variable stores an integer (there are four `int` constants, one for each possible direction) which indicates the direction in which the `CoolCat` object should move when it is told to move (i.e., when the `move()` instance method is called). Initially the `CoolCat` object should be ready to move either upwards or downwards (your choice).

Initially the `speed` of the `CoolCat` object is a random number (mine is 4, 5, 6, 7 or 8).

The `CoolCat` object moves up, down, right or left depending on its current direction.

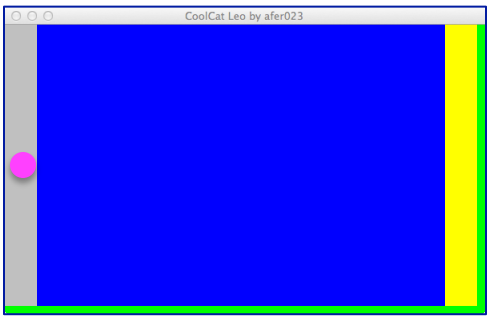
The `move()` method moves the `CoolCat` object in its current direction. Note that the `CoolCat` object should never be visible outside the whole game area (the outside border of the three game areas).

The `hasReachedHome()` method returns `true` if the rectangle passed to this method as a parameter intersects the area of the `CoolCat` object. Otherwise this method returns `false`.

The code on Page 54 (example 33) of your Code Examples booklet can be used as a guide (this is just a guide, not the correct code) to help you define the `CoolCat` class.

In the `A3JPanel` class define an instance variable of type `CoolCat`, create the `CoolCat` object (in the constructor) and draw the `CoolCat` object (by calling the `draw()` method in the `CoolCat` class in the `paintComponent()` method).

Run the `A3` application and check that the `CoolCat` object is visible and is initially positioned correctly.



Stage 4 – Handling KeyEvents

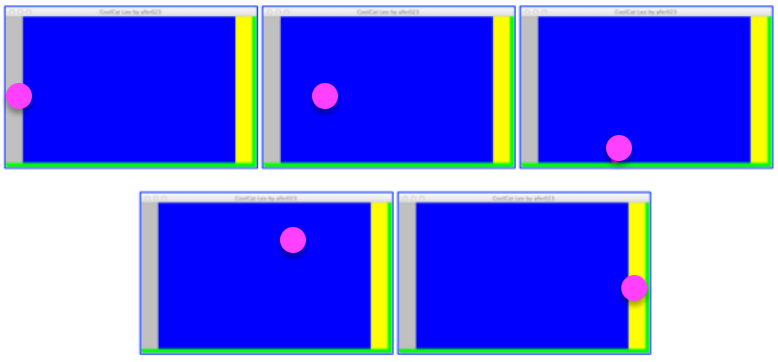
Add code to the `A3JPanel` class so that the JPanel responds to `KeyEvents`. In the `keyPressed()` method add code to change the direction of the `CoolCat` object whenever the user presses one of the arrow keys. Note that currently the `CoolCat` object does not move so you will not see the effect of your code for this stage until you have completed Stage 5. There is an example of a JPanel which handles KeyEvents on Page 50 (example 30) of your Code Examples booklet.

Stage 5 – Implementing a Timer

The `CoolCat` object moves in its current direction with every tick of the timer. Add code to the `A3JPanel` class so that the JPanel responds to `ActionEvents`. In the `A3JPanel` class declare a `Timer` object as an instance variable and construct the `Timer` object. To facilitate testing, start the `Timer` object ticking at the end of the constructor method. This will allow you to check that the `CoolCat` object moves correctly in all four directions. In the `actionPerformed()` method add code to tell the `CoolCat` object to 'move' with every tick of the timer.

Again the code on Page 55 (example 33) of your Code Examples booklet can be used as a guide (this is just a guide, not the correct code) to help you with the implementation of the `Timer` object.

Check that the `CoolCat` object does not move outside the game area. Check that the `CoolCat` object changes to the desired direction whenever the user presses one of the four arrow keys (Stages 3 and 4).



Note: the CoolCat is abnormally large in these screenshots.

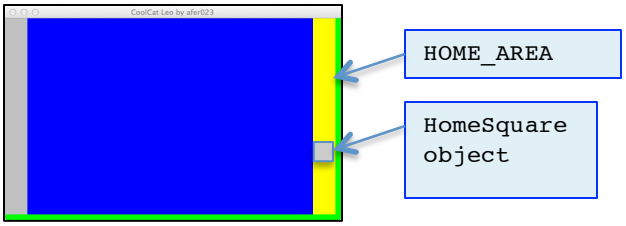
Stage 6 – Define the HomeSquare class

A `HomeSquare` object is a rectangular object which moves up and down the area defined by the constant, `HOME_AREA`. Later in the assignment the player will try to reach the home square without running into any of the sliding walls.

For this stage of the assignment you need to define the `HomeSquare` class. The skeleton of the `HomeSquare` class is shown below:

```
public class HomeSquare {
   private Rectangle area;
   private int speed;
   private int direction;

   public HomeSquare() { /* constructor */  }
   public Rectangle getArea() { ... }
   public void changeDirection() { ... }
   public void changeSpeed() { ... }
   public void move() { ... }
   public void draw(Graphics g) { ... }
}
```



Things to note about the HomeSquare class

The `HomeSquare` class has 3 instance variables.

The `area` instance variable represents the current position, width and height of the `HomeSquare` object.

Initially the `HomeSquare` object is positioned in the area given by the constant, `HOME_START_AREA`.

The `direction` instance variable stores an integer (two `int` constants, `UP` and `DOWN` have been defined) which indicates the direction in which the `HomeSquare` object will move (when it is 'told' to move). Initially the `HomeSquare` object should be ready to move either upwards or downwards (your choice).

Initially the speed of the `HomeSquare` object is a random number (mine is 2, 3, 4, 5 or 6).

The `changeSpeed()` method sets the value of the `speed` instance variable to a random number (mine is 2, 3, 4, 5 or 6).

The `HomeSquare` object moves upwards or downwards depending on its current direction. Note that the `HomeSquare` object should never be visible outside the home area (defined by the `HOME_AREA` constant). You may wish (this is optional) to change the direction of the `HomeSquare` object when it reaches the top or the bottom of the home area.

The `changeDirection()` method changes the direction of the `HomeSquare` object, i.e., if it is currently moving upwards the direction is set so that the `HomeSquare` object moves downwards and if it is currently moving downwards the direction is set so that the `HomeSquare` object moves upwards. As well, whenever the `HomeSquare` object changes its direction it also is assigned a new random speed (in the `changeDirection()` method make a call to the `changeSpeed()` method).

In the `A3JPanel` class define an instance variable of type `HomeSquare`, create the `HomeSquare` object and draw the `HomeSquare` object (by calling the `draw()` method in the `HomeSquare` class in the `paintComponent()` method). In the `actionPerformed()` method add code to 'tell' the `HomeSquare` object to 'move' with every tick of the timer. Also add code so that with every tick of the Timer there is a 5% (or some other percentage of your choice) chance that the `HomeSquare` object is 'told' to change its direction.

Run the `A3` application and check that the `HomeSquare` object is visible and is initially positioned correctly. Check that the object moves correctly up and down the home area and randomly changes direction and speed. Check that the `HomeSquare` object is never visible above or below the home area.

Stage 7 – Define the SlidingWall class

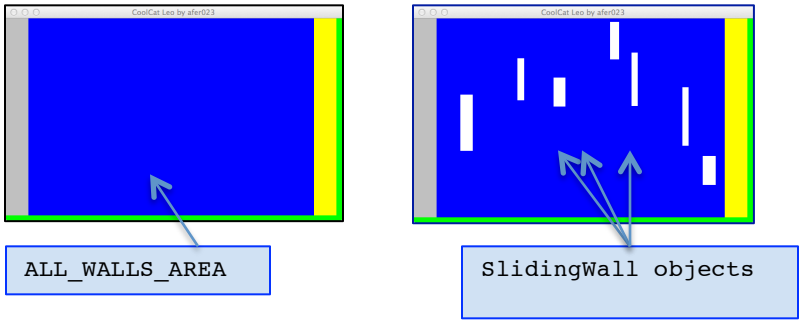
A `SlidingWall` object is a rectangular object which moves up or down the middle part of the game area defined by the constant, `ALL_WALLS_AREA`. Later in the assignment the player will try to reach the home square without running into any of the sliding walls.

For this stage of the assignment you need to define the `SlidingWall` class. The skeleton of the `SlidingWall` class is shown below:

```
public class SlidingWall {
   private Rectangle area;
   private int speed;
   private boolean isVisible;
   private int direction;

   public SlidingWall() { /* constructor */  }

   public boolean getIsVisible() { ... }
   public Rectangle getArea() { ... }
   public boolean intersectsTheWall(Rectangle leoArea) { ... }
   public void move() { ... }
   public void draw(Graphics g) { ... }
}
```



Things to note about the SlidingWall class

The `SlidingWall` class has 4 instance variables.

The `area` instance variable represents the current position, width and height of the `SlidingWall` object.

The `direction` instance variable stores an integer which indicates whether the `SlidingWall` object is moving upwards or whether it is moving downwards. Initially the `SlidingWall` object has an equal chance of moving upwards or moving downwards.

Initially the `SlidingWall` object is positioned in a random position either exactly above the `ALL_WALLS_AREA` if it is moving downwards or exactly below the `ALL_WALLS_AREA` if it is moving upwards. The width and height of the sliding wall is random (use the constants `MAX_SLIDE_HEIGHT`, `MIN_SLIDE_HEIGHT`, `MAX_SLIDE_WIDTH` and

`MIN_SLIDE_WIDTH` to obtain suitable numbers for the width and height). The horizontal position of the object is random but make sure that the sliding wall is never visible (horizontally visible) outside the walls area defined by the constant, `ALL_WALLS_AREA.`

Initially the speed of the `SlidingWall` object is a random number (mine is 12, 13, 14, 15 or 16).

The `move()` method moves the `SlidingWall` object upwards or downwards depending on its direction.

The `intersectsTheWall()` method returns `true` if the rectangle passed to this method as a parameter intersects the wall area. Otherwise this method returns `false`.

Initially the `isVisible` instance variable should be set to `true`. If a wall is moving upwards it should stop being visible once it has completely moved above the top of the `ALL_WALLS_AREA` area. If a wall is moving downwards it should stop being visible once it has completely moved below the bottom of the `ALL_WALLS_AREA` area.

You will not be able to test this code until you have completed the next two stages of the assignment.

Stage 8 – Define the WallsManager class

The `WallsManager` class handles an array of sliding walls. The skeleton of the `WallsManager` class is shown below:

```
public class WallsManager {
   private SlidingWall[] walls;
   private int numberOfWalls;

   public WallsManager() { /* constructor */ }

   public void addAWall() { ... }
   public boolean checkIfLeoAreaIntersectsAWall(Rectangle leoArea) {
                                                                  ... }
   public void moveTheWalls() { ... }
   public void drawTheWalls(Graphics g) { ... }
}
```

Things to note about the WallsManager class

The `SlidingWall` class has 2 instance variables.

The `walls` instance variable is an array of `SlidingWall` objects. The constant, MAX_WALLS is the number of elements in this array. Initially there are no `SlidingWall` objects in the array, i.e., all the elements are `null`.

The `numberOfWalls` instance variable stores an integer which indicates how many `SlidingWall` objects are currently in the `walls` array. Initially this is set to the value 0.

The `addAWall()` method creates a new `SlidingWall` object and adds it to the `walls` array. This should only be done if there is room in the array. As well the method increases the `numberOfWalls` instance variable.

The `moveTheWalls()` method moves all the `SlidingWall` objects which are currently in the `walls` array.

The `checkIfLeoAreaIntersectsAWall()` method returns `true` if the rectangle passed to this method as a parameter intersects any of the `SlidingWall` objects in the `walls` array. Otherwise this method returns `false`. This method will need to check each element of the `walls` array (don't forget that the `SlidingWall` class has an instance method, `intersectsTheWall()`).

You will not be able to test this code until you have completed the next stage of the assignment.

Stage 9 – The WallsManager instance

In the `A3JPanel` class define an instance variable of type `WallsManager`, create the `WallsManager` instance and draw the walls by calling the `drawTheWalls()` method on the `WallsManager` instance. Currently when you run the `A3` application you will not see any sliding walls in the JPanel because initially there are no sliding walls in the `WallManager`'s in the `walls` array. In the `actionPerformed()` method add code to tell the `WallsManager` object to 'moveTheWalls' with every tick of the timer. Also add code so that with every tick of the timer there is a 10% (or some other percentage) chance that the `WallsManager` object is 'told' to add a new sliding wall to its array (by calling the `addAWall()` instance method).

Run the `A3` application and check that sliding walls randomly appear and move either up or down the middle section of the game area. Check that the sliding walls are randomly positioned and have random widths and heights and that they are never created so that they are horizontally visible outside the `ALL_WALLS_AREA` area.

Stage 10 – Tidying up the SlidingWall array

In the `A3JPanel` class you will notice that any sliding walls which are moving downwards become visible below the `ALL_WALLS_AREA` area. Also any sliding walls which are moving upwards are initially visible below the `ALL_WALLS_AREA` area. The constant, `EXITING_SLIDES_AREA`, defines a rectangular area below the game area. In the `paintComponent()` method fill this rectangular area with the same colour as your game screen colour (use the `GAME_SCREEN_COLOUR` constant). This will cover any sliding walls which are either starting below the game area or exiting the game area.

In the `WallManager` class the array is big enough to hold 50 (given by the `MAX_WALLS` constant) `SlidingWall` instances. After all the walls in the array have moved, add code which removes (from the `walls` array) any walls which are completely outside the `ALL_WALLS_AREA` area, i.e., any sliding walls which are no longer visible (their `isVisible` instance variable has been set to `false` - in the `SlidingWall` class). Define two helper methods to help carry out this task:

```
            //Goes through the SlidingWall array BACKWARDS and
            //checks if a wall is not visible.  Any wall which is not visible is removed
            //from the array i.e., the other walls are moved one position down the
            //array and the number of walls currently in the array is reduced by 1.
private void removeInvisibleWalls() { ... }
            //move the walls from the index position given by the parameter
            //one position down the array, i.e., first element to move is fromIndex + 1
private void moveWallsDownArray(int fromIndex) { ... }
```

Run the A3 application and check that sliding walls behave correctly. Make sure there are no NullPointerExceptions when you run your application. Note that you should not change the value of the MAX_WALLS constant.

Stage 11 – Start/stop the Timer correctly

Up to this stage, for testing purposes, the timer starts ticking as soon as the JPanel is displayed. Set up the JPanel to handle MouseEvents. Pressing the mouse should start/stop the timer, i.e., if the timer is stopped then the timer should start when the user presses the mouse, and if the timer has started then the timer should stop when the user presses the mouse. There is an example of a JPanel which handles MouseEvents on Page 48 (example 28) of your Code Examples booklet.

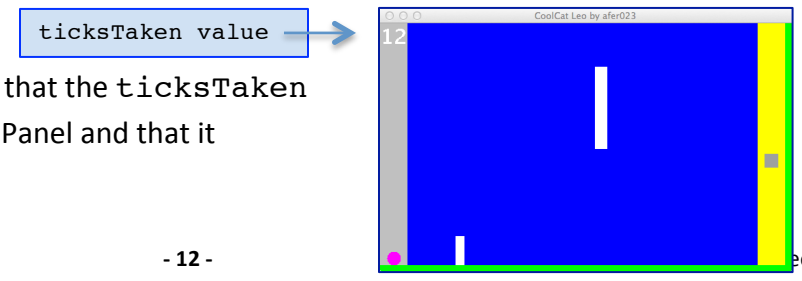
Run the A3 application and check that the timer does not start (i.e., there is no animation) until the user presses the mouse. Also check that the user can stop/start the timer by pressing the mouse.

Stage 12 – Timing the CoolCat game

We now want to set a time limit for the game. In the A3JPanel class define two instance variables:

```
private int ticksTaken;
private int tickCounter;
```

These variables will be used to time the game and initially they are both set to 0. With every tick of the timer the tickCounter instance variable should increase by 1. When the tickCounter instance variable reaches 15 (or whatever you decide) increase the ticksTaken instance variable by 1 and reset the tickCounter back to 0. When the ticksTaken instance variable reaches 30 (given by the TICKS_ALLOWED constant - or a number of your choice) the timer will stop (see next stage). The value in the ticksTaken instance variable is displayed in a colour of your choice in the top left position on the screen (use the Point constant, TICKS_POSITION) to position the value of the ticksTaken instance variable.



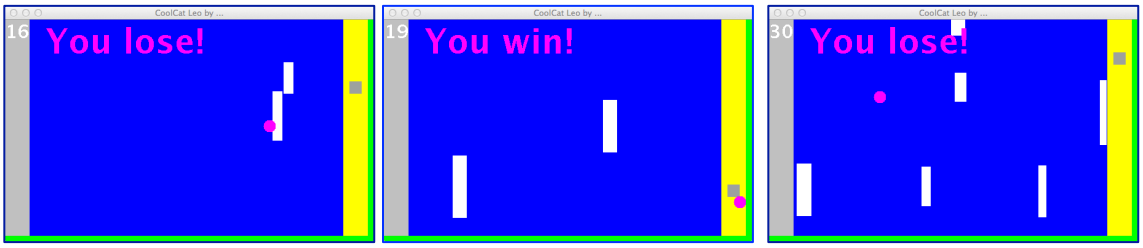
Run the A3 application and check that the ticksTaken is displayed in the top left of the JPanel and that it

increases.

Stage 13 – End of game

To store the end of game situation, in the `A3JPanel` class define two instance variables (initially they are both `false`):

```
private boolean leoHasWon;
private boolean gameHasEnded;
```



The game finishes (and the timer stops) either

when the `CoolCat` object runs into one of the sliding walls, `CoolCat` loses

or

when the `CoolCat` object reaches (intersects) the `HomeSquare` object on the right hand side of the game area. `CoolCat` wins.

or

when the time is up. `CoolCat` loses.

The end of game is checked at the end of the `actionPerformed()` method after all the game objects have moved. Define a helper method:

```
private boolean setResultAndGetEndOfGame () { ... }
```

and make a call to this helper method at the end of the `actionPerformed()` method. This helper method returns `true` if the game has ended, otherwise the method returns `false`. As well as returning `true` or `false` the method sets the `leoHasWon` instance variable to `true` or `false`. The method does three checks:

1. The method checks if the `CoolCat` object intersects any of the sliding walls by calling the `checkIfLeoAreaIntersectsAWall`() method (using the `WallManager` instance). The parameter passed to this method is the area of the `Coolcat` instance. If the `CoolCat` object intersects any of the sliding walls the game has ended and the `CoolCat` has lost.

2. The method checks if the `CoolCat` object intersects the `HomeSquare` object by calling the `hasReachedHome()` method (using the `CoolCat` instance). The parameter passed

to this method is the area of the `HomeSquare` instance. If the `CoolCat` object intersects the home square the game has ended and the `CoolCat` has won.

3. The method checks whether the `ticksTaken` instance variable reaches 30 (given by the `TICKS_ALLOWED` constant). If the `ticksTaken` instance variable reaches 30 the game has ended and the `CoolCat` has lost.

If the call to the `setResultAndGetEndOfGame()` helper method returns `true`, the timer stops and the `gameHasEnded` instance variable is set to `true`. If the `gameHasEnded` instance variable is `true` the `paintComponent()` method of the JPanel should display a message stating whether the player has won or lost ("You win!", "You lose!" is adequate). The message should be displayed in a colour of your choice in the position given by the `Point` defined by the constant, `WINNER_LOSER_INFO_POSITION`. Again use a helper method to do this:

```
private void displayWinnerLoser(Graphics g) {  ... }
```

Once the game has ended pressing the mouse should not start the timer again. You may decide to allow the user to reset the game as your extra feature (see stage 16).

Stage 14 – Add a title screen

When the program first starts, the user should be presented with a simple title screen displaying the name of the game (your choice), your login name, what stage of the assignment you have completed, any instructions you may wish to tell the user (mention anything in your assignment which works in a different way) and a prompt asking the user to press the mouse to see the game screen and telling the user to press the mouse again in order to start the game.

After you have completed Stage 16, your title screen should also contain a description of the feature which you have added to your game.

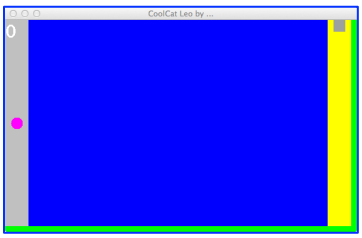
An example of a title screen is shown below.

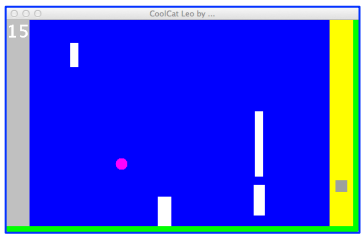
Stage 15 – Changing from title screen to game screen

When the user presses the mouse, the title screen disappears and the game screen is displayed.

Add code to the `JPanel` class which makes the program initially display the title screen, then display the game screen when the user first presses the mouse. Use a boolean instance variable, `isShowingTitleScreen`, to indicate whether the title screen is currently showing and set this boolean to `false` when the game screen should be displayed (don't forget `repaint()`). Because the animation should not start when the screen changes, you will need to make use of the `isShowingTitleScreen` boolean in the `actionPerformed()` method.

Important. When the user first presses the mouse, the game screen is showing but the animation should not commence, i.e., nothing should move until the user presses the mouse a second time to start the timer.





Stage 16 – Add your own feature to the game

Add ONE feature of your own to your game. In the title screen (see Stage 14) you must display a description of your added feature. Example features you may add are:

- any feature you wish,
- allow the user to reset the game by doing something
- have other animation happening (e.g., crash, boom!)
- display some more interesting game statistics
- make game levels
- add some Angel objects which help the player
- a cheat feature
- make the sliding walls never intersect each other (see suggested helper methods below).

IMPORTANT TO NOTE: the parameters which are passed to the `SlidingWall` constructor are different if you are implementing this as your feature.

```
public class SlidingWall {
  ...
  public SlidingWall(SlidingWall[] allWalls, int numberOfWalls) {
                                                  /* constructor */  }
  private int getRandomX(int wide) { ... }
  private int getRandomHeight() { ... }
  private int getRandomWidth() { ... }
  private void xOverlapsOneOfTheWallAreas(SlidingWall[] allWalls,
                                          int numberOfWalls) { ... }
}
```

Academic honesty (this is very important!)

This assignment is an **assessed piece of coursework**, and it is essential that the work you submit reflects what you are capable of doing. You must not copy any source code for this assignment and submit it as your own work. You must also not allow anyone to copy your work. All submissions for this assignment will be checked, and any cases of copying/plagiarism will be dealt with severely. We really hope there are no issues this semester in CompSci 101, so please be sensible!

Ask yourself:

"Have I written the source code for this assignment myself?"

If the answer is "no", then **please talk to us before the assignments are marked**.

Ask yourself:

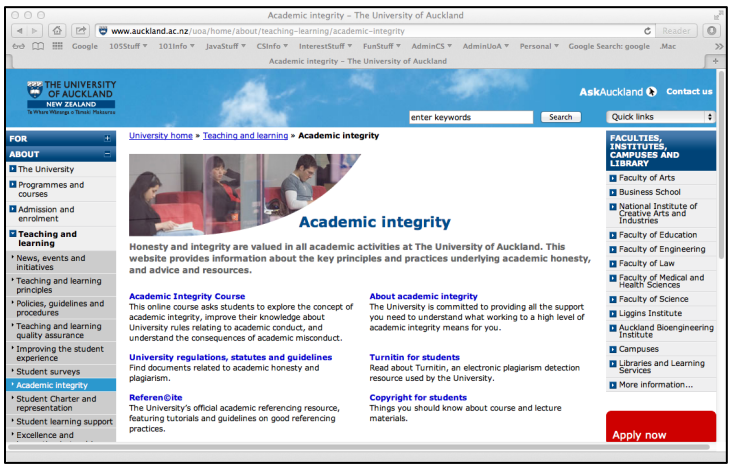
"Have I given *anyone* access to the source code that I have written for this assignment?"

If the answer is "yes", then **please talk to us before the assignments are marked**.

Once the assignments have been marked it is too late.

There is more information regarding The University of Auckland's policies on academic honesty and plagiarism here:

`http://www.auckland.ac.nz/uoa/home/about/teaching-learning/honesty`



Submission

You have worked hard on this assignment, and we want to make sure that you are rewarded for your effort.

You should submit **SEVEN JAVA SOURCE** files for this assignment (there is the option to submit your own extra java classes) and one text file (see below):

- `A3.java`
- `A3JFrame.java`
- `A3JPanel.java`
- `CoolCat.java`
- `HomeSquare.java`
- `SlidingWall.java`
- `WallsManager.java`
- Any other Java file you have defined
- `A3.txt` (see below for the contents)

Do not submit your .class files.
IMPORTANT: **check that the code you submit compiles.** If your code has syntax errors of any kind, the compiler will tell you what they are – correct all syntax errors before submitting your code.

It will be much better for you to submit code that compiles but is not functionally complete than to submit code which does not compile.

Submit your file using the Web Drop Box (https://adb.ec.auckland.ac.nz/adb/). This link is available on the CompSci 101 website (Assignments page).

Submission

You **must** include a text file named A3.txt in your submission. There will be a 5 mark penalty for not doing so. This text file must contain the following information:

Your full name
Your login name and ID number
How much time did the assignment take overall?
What areas of the assignment did you find easy?
What areas of the assignment did you find difficult?
Any other comments you would like to make.

Marking

Style – 10 marks

Comment at the top of each class (containing your name, upi, date and a brief description of the class), good variable names, good method names, correct indentation, uses the constants provided and uses helper methods to simplify and structure the code.

Correctness – 90 marks

Stage 1 – Your UPI – 4 marks
Stage 2 – Display the game area – 5 marks
Stage 3 – Define the `CoolCat` class – 5 marks
Stage 4 – Handling `KeyEvents` – 5 marks
Stage 5 – Implementing a `Timer` – 5 marks
Stage 6 – Define the `HomeSquare` class – 5 marks
Stage 7 – Define the `SlidingWall` class – 8 marks
Stage 8 – The `WallsManager` class – 9 marks
Stage 9 – The `WallsManager` instance – 5 marks
Stage 10 – Tidying up the sliding walls – 5 marks
Stage 11 – Start/stop the `Timer` correctly – 5 marks
Stage 12 – Timing the CoolCat game– 6 marks
Stage 13 – End of game – 8 marks
Stage 14 – Add a title screen – 5 marks
Stage 15 – Changing from title screen to game screen – 5 marks
Stage 16 – Add your own feature to the game – 5 marks

Penalty of 15 marks if the program does not use `Rectangle` objects to store the position and size of the game objects (`CoolCat`, `HomeSquare`, `SlidingWall`).

Penalty of 5 marks Defines extra variables as instance variables where the variables should be defined as local variables.

Penalty of 5 marks A3.txt information was not completed