Aim of the coursework:

You will implement a memory/matching game in Java. You can see a demo of it linked from the moodle page.

The game has a grid of buttons, which represent cards that you can turn over. Initially all of the buttons show blank/grey so you don’t know what colour they are underneath. You will turn over two, one at a time, and if they match you get the points.

It also has labels at the top and bottom which specify whose turn it is, the score for the player and what they should do.

A player will select one button then another. As they select the buttons, they button will reveal its colour.

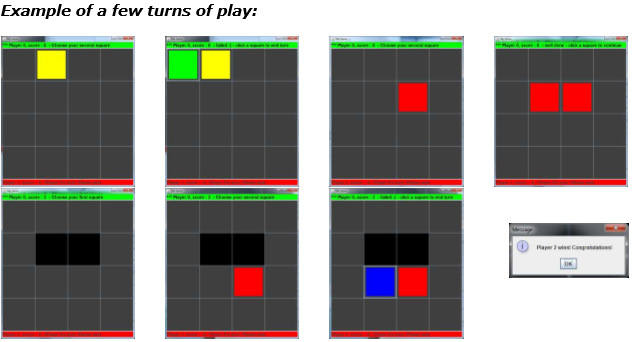
If the player selects two buttons of the same colour, the game will tell them, then when they click any button again (to continue) the two buttons that they got correct will turn black and the player has another attempt to choose two buttons.

The player may not choose the same button twice and may not choose a black button.

If the player chooses two buttons which are different colours then again the game tells them and when they click again it becomes the other player’s turn.

When all buttons are black, the game tells the players who won (or that it was a draw) and when they press OK on the message it starts again.

At all times the game tells the players what is happening by using two labels – one for player 0 at the top of the window and one for player 1 at the bottom of the window. The message for the active player should have a green background and the message for the other player should have a red background.



Assessed requirements: Note that the earlier requirements are deliberately more precise/step-by-step than later ones. You will probably want to refer to the hints more for later requirements.

1. Create a main class and name it using your capitalised username with a Main at the end. E.g. my username is pszja so I would call mine PszjaMain (note that classes should always have a capital letter as the first letter).

2. Create a main() function and ensure that your main function does only one thing: create an instance of the class. If you don’t want to do the work in the constructor, you MAY (if you wish) immediately call a single method on the new object but you should NOT create an object reference variable to store a reference to the created object (it is not necessary). If your finished code works properly, it should be possible to create two instances of the class and have two completely separate copies of the game running at once – as I did in the demo video. None of the other methods of your main class should be static.

3. Create five ‘final static int’ data members/attributes for your main class:

i. WIDTH – the number of buttons wide that your display is

ii. HEIGHT – the number of buttons high that your display is

iii. COLOURS (you can use the US spelling COLORS if you wish) – the number of different colours you will have for your buttons

iv. MAX\_PER\_COLOUR – the number of buttons you will have of each of the colours

v. BUT\_SIZE – the size (in pixels) of each button

These will be constants (final means you cannot change the value) and should ensure that you use these names rather than the actual numbers in your code for this class. E.g. to create a button make it’s width equal to BUT\_SIZE rather than 100, for example. You create these and should use them appropriately throughout your code rather than hard-coding numbers. Note: You should always set these values so that WIDTH\* HEIGHT = MAX\_PER\_COLOUR \* COLOURS, and MAX\_PER\_COLOUR is an even number (so that you can pair up squares). Your program does not need to verify this (assume that it will never be set to an odd number). We will want to see it run with a 4 by 4 grid with 4 colours and 4 squares per colour, as well as with and 8 by 8 grid with 8 colours.

4. Create a buttonClicked( int iButton ) method (not static!) in your main class (you can change the name/parameter name if you wish).

5. Create a ColorButton class which is a sub-class of JButton. Add a constructor which takes at least four parameters, specifying a number for the button (and int), a reference to your main class object and the minimum/preferred width and height for the button. Set the size appropriately and store the main object reference and the button number in member variables (see step 10 for why).

6. Your buttons represent the card in a matching game. They can have three states – showing the back of the card (grey), showing the front of the card (i.e. turning the card over to see what colour it is, in which case the buttons shows the specified colour) or finished (the equivalent of taking the cards away because they were matched – in which case make them black). Add oneor more variables to manage the state of a button (showing the front/showing the colour, showing the back/showing grey, or taken away/already completed/showing black. Initialise this state for each button to showing the back of the card (grey).

7. Add a method to the new button class called flip() which has a Boolean parameter specifying whether the button should show its colour or not. Calling flip(true) should make the button show its front (i.e. show the hidden colour) and calling flip(false) should make the button show its back (i.e. grey).

8. Add a variable to store the real button colour. Add a set… method to allow this variable’s value to be set when a colour is assigned to the button.

9. Implement the appropriate draw method of the button class to draw the button in the appropriate colour depending upon the state of the button: black if the button has been completed, grey if it has not been selected, and the appropriate colour if it has been selected. 10. Add a listener to the button, so that the button itself knows when it has been pressed and calls the buttonClicked() method of the main class, telling it which button number was clicked.

At this point I tested my ColorButton class before continuing.

Now return to your main class.

As you implement the code, remember to use the constants which you created in step 3. You may use one main class or create additional classes to implement features if you wish – it is up to you.

11. Implement your main program to create a JFrame.

12. Add two labels at the top and bottom of your frame – these will display the status of the players and tell them their score and what to do. Give the labels appropriate fonts and background colours.

13. Implement appropriate variables in the main object to maintain the current state. E.g. you will need to know the score for each player, whose turn it is, what is currently happening (e.g. is the current player about to select the first square, a second square or to click a button to say to continue after succeeding or failing). You may need to implement other variables as well, either on the main object or on the buttons objects – feel free to do so.

14. Implement a method which will consider the current state and display an appropriate message on the labels telling the players what to do.

15. Set appropriate background colours on each label so that the label for the current player is green and for the other player is red.

16. Ensure that the current score for each player is correctly shown on the label for the player. 17. Add a grid of WIDTH x HEIGHT ColorButtons to the centre of the frame and give each a different number so that you can identify which was pressed.

18. Randomly allocate colours to the buttons.

19. Ensure that the number of different colours is equal to the COLOURS constant.

20. Ensure that there are MAX\_PER\_COLOUR buttons of each colour.

21. A player should be able to choose a first square and it should be revealed when they select it. The player should not be able to choose a finished (i.e. black) square (any click on a black square should be ignored). The game should prompt them to choose a second square at this time.

22. A player should be able to choose a second square and it should be revealed when they select it. They cannot choose the same square as for the first square, or a black square (if they do so then ignore the click so that they can choose another one).

23. After choosing a second square, the game should check whether they found a match. If they did, the player should be told so in the label and be prompted to click any square to continue. If they did not match then the player should be told so and prompted to click a square to continue.

24. After clicking to continue, if the player matched the colours of the two squares, turn both black, and the player receives 2 points.

25. After correctly matching squares and the squares going black, the player be given another go and should be prompted to choose a new first square.

26. If the player failed then the game should turn the squares back to grey so that neither player can see their colour by calling the flip(false) method on each of the two buttons. The game should then move to the other player and allow them to choose a first square, continuing the game.

27. If all of the squares have turned black then the game should display a message window (e.g. using JOptionPane) saying who the winner is – either one of the players or a draw based on who has the most points.

28. After the message saying who the winner is, the state should reset allowing another game to be played, with new random colours for the buttons.

29. For the last two marks, implement an iterator pattern so that some class supports the Iterable interface and produces an Iterator which allows you to iterate through the buttons. You could use this, for example, to see whether all of the buttons had been clicked, or you could just use it for a test in some test code that you call at some point.

30. This mark is for your Iterator for your iterator pattern demonstrably working correctly.

Test your game fully. You may find it useful to reduce the number of buttons to 2x2 and colours to 2 to do fast testing of things like the final message being displayed when a player wins.