CSC 15 Spring 2015
Project #2
Due Date: Friday May 5th by midnight
Email your project to: Csc15grader@gmail.com

Memory Matching Game

A common memory matching game played by children is to start with a deck of cards that contains identical pairs. For example, given six cards in the deck, two might be labeled 1, two might be labeled 2 and two might be labeled 3. The cards are shuffled and placed face down on the table. A player selects two cards that are face down, turns them up, and if the cards match they are left face up. If the cards do not match they are returned to their original face down position. The game continues until all the cards are face up. The goal is to get all cards face up in as few turns as possible.

The problem

Write a program that plays the memory matching game. Your program must have three different classes:

- **Card** class encapsulating the concept of the card
- **MemoryGameBoard** class encapsulating the concept of the memory game board.
- **playMemoryGame** class that is the application that will control the play of the game using the methods from the MemoryGameBoard class. **This is the client/ tester class.**

UML diagram for the Card class

Card
-value: int -faceUp: boolean
+Card(initValue: int) +getValue(): int +isFaceUp() : boolean +flipCard() : void +setFaceUp(): void +setFaceDown(): void +equals(otherCard: Card): boolean +toString(): String

Instance variables

value: An integer variable representing the value of the card
faceUp: A boolean variable indicating if the card is face up or down

Method descriptions

- **Card(int initValue) :** Explicit value constructor initializes **value** to initValue and **faceUp** to false.

- **getValue():** Returns the value of the instance variable **value.**
- **isFaceUp():** Returns the value of the variable **faceUp**.

- **flipCard() :** Reverses the value of **faceUp.** For example, if faceUp is false, it becomes true.

- **setFaceUp():** Sets **faceUp** to true.

- **setFaceDown():** Sets **faceUp** to false.

- **equals(Card otherCard):** Returns true if the value of this card is equal to the value of otherCard, otherwise it returns false.

- **toString():** Returns a String that contains the value of the Card if the card is face-up or an asterisk if the card is face-down.

UML diagram for the MemoryGameBoard class

MemoryGameBoard
+BOARD_SIZE : int = 4 -NUM_SWAPS : int = 1000 -gameBoard: card[][]
+MemoryGameBoard() +shuffleCards() : void +showBoard(): void +cardsMatch(row1: int, col1: int, row2: int, col2: int): boolean +isFaceUp(row: int, col: int) : boolean +flipCard(row1: int, col1: int) : void +toString(): String -initalizeCards(): void -hideBoard():void -turnCardsFaceDown() : void +displayGameBoardValues(): void (This method is to be used only for testing purposes while the program is under construction.)

MemoryGameBoard class

Constants

- **BOARD_SIZE** represents the dimensions of the portion of the **gameBoard** array of **Card** objects that the memory game will use. (See the description of the gameBoard instance variable.)
- **NUM_SWAPS** represents the number of times that pairs of Cards in the **gameBoard** array need to be swapped to shuffle the cards.

Instance variables

- **gameBoard** A (**BOARD_SIZE+1) X** (**BOARD_SIZE+1)** array of **Card** objects representing cards used in the memory game. NOTE: The dimensions of the array are one more than BOARD_SIZE because the zero row and zero column in the array will be ignored, i.e. the memory game will not use them.

Method descriptions

- **MemoryGameBoard()**
  No-argument constructor instantiates and initializes the **gameBoard** array, and then it calls the method **initializeCards()** to instantiate and initialize the individual Cards in the gameBoard array.

- **shuffleCards()**
  Calls the method **turnCardsFaceDown.**
  Repeatedly (NUM_SWAPS times) selects two cards at random locations and swaps them**.**
  This method requires:
  - a loop that repeats NUM_SWAPS times
    - generates 4 different random numbers for row1, col1, row2, col2 (numbers must be in the range 1… BOARD_SIZE)
    - swaps the Card in the gameBoard array at the location [row1][col1] with the Card at the location [row2][col2]

  The table below shows what the array of cards might look like after its elements have been shuffled.

```
       1 2 3 4
<<<   =========
<<<1 | 3 4 8 2 |
<<<2 | 5 6 5 1 |
<<<3 | 3 7 4 7 |
<<<4 | 6 2 1 8 |
<<<   =========
```

- **showBoard()**
  Calls the method **hideBoard()** to hide the previously printed board**.**
  Uses the toString method to display the current state of the ***gameBoard*** on the monitor.

- **cardsMatch**(int row1, int col1, int row2, int col2)
  Returns true if the card at **gameBoard[row1][col1]** is equal to the card at **gameBoard[row2][col2]** otherwise false is returned. NOTE: Use the **equals** method in the **Card** class.

  Example: For the gameBoard shown on the previous page, **cardsMatch(1, 1, 1, 3)** will return true because the value of the card at **gameBoard[1][1]** is equal to the value of the card at **gameBoard[1][3]**.

  NOTE: use the **equals** method in the Card class.

- **isFaceUp(int row, int col)**
  Returns the value of the instance variable **faceUp** for the Card at **gameBoard[row][col].** This method will call the method isFaceUp() in the Card class.

- **flipCard(int row, int col)**
  Turns over the Card at **gameBoard[row][col]** by calling the **flipCard** method in the Card class.

- **toString()**
  Builds and returns a String that contains the current state of the **gameBoard** in the format shown below. It goes through the **gameBoard** array using the **toString()** method from the Card class to add each of the cards to the String.

  The table below shows the output from **toString()** at the beginning of the game before any matches have been found.

```
        1  2  3  4
      =========
<<<1 | * * * * |
<<<2 | * * * * |
<< 3 | * * * * |
<< 4 | * * * * |
<<<    =========
```

  The table below shows the output from **toString()** after one match has been found.

```
     1 2 3 4
<<<    =========
<<<1 | 3 * * * |
<<<2 | * * * * |
<<<3 | 3 * * * |
<<<4 | * * * * |
<<<    =========
```

- **initializeCards()**
  Instantiates and initializes the cards in the used portion of the **gameBoard** array such that the values of the cards are pairs of numbers 1 through 8 in the locations shown in the table below.

```
        1 2 3 4
<<<    =========
<<<1 | 1 2 3 4 |
<<<2 | 5 6 7 8 |
<<<3 | 1 2 3 4 |
<<<4 | 5 6 7 8 |
<<<    =========
```

- **hideBoard()**
  Hides the last board printed on the screen by scrolling it down using System.out.println() .For example to scroll down 10 lines repeat System.out.println() 10 times.

- **turnCardsFaceDown()**
  Sets the value of the instance variable **faceUp** to **false** for all the Cards in the **gameBoard** array by calling the method **setFaceUp** in the **Card** class.

- **displayGameBoardValues()**

NOTE: This method is to be used only for testing purposes. This method will display the value of the cards in the gameBoard array. You can call this method in the application during testing. This will allow you to see where the cards are so you can find the matches easily while you are testing. You would of course want to remove all calls to this method once the program is finished and works well.

PlayMemoryGame class

This program will keep track of how many turns it takes for the player to find all the matches.

It will also allow the player to play as many times as he/she wishes and it will keep track of the player's best score.

Declare two static variables before the main method:

- A Scanner object to read from the keyboard
- An object of the **MemoryGameBoard** class called **game.**

NOTE: You are not allowed to create any more static variables outside the main method.

public static void main (String[] args)

- Declare a variable called ***bestScore*** to keep track of the player's best score. The score represents the number of turns it takes to find all the matches, so the smaller the score the better. To start with initialize this score to some arbitrarily large number that you know the player will beat the first time the player plays. (Maybe 500, I know the player will do better than that.)
- Declare the variable ***turnCount*** to keep track of how many turns it takes for the player to find all the matches.
- Declare a variable called ***pairsLeft*** to keep track of how many matching pairs there are still to be found.
- That is good for now. You can come back here and declare other variables as you need them.

- Print headings and the goal of the game

- Loop while the player wants to start a new game
  - Shuffle the cards.
  - Initialize ***turnCount*** to zero.
  - Initialize ***pairsLeft*** to $(\text{BOARD\_SIZE})^2 / 2$
  - Loop while there are still matching pairs to find
    - Add 1 to the turn count
    - Display the board
    - Call the method `getValidInt` to get values for row1, col1
      - While the card at the chosen location is face-up, call the getValidInt method to get another row1, col1

    - Flip the card at row1, col1
    - Call the method `getValidInt` to get values for row2, col2
      - While the card at the chosen location is face-up, call the getValidInt method to get values for row2, col2
    - Flip the card at row2, col2
    - Display the board
    - If the selected cards match

            - print a message that you found a match.
            - Update **pairsLeft** appropriately.
          - Otherwise
            - flip the cards at locations row1, col1 and row2, col2.
            - Print a message about no match was found.
      - output **turnCount**
      - adjust bestScore as appropriate
      - print bestScore
      - print a message that game is over
      - Ask the user if he/she wants to play again.
  - Print a goodbye message

public static int getValidInt(String prompt, int min, int max)

Arguments

prompt: string to prompt the user – (For this program the client will pass either "   row: " or "   col: " to the prompt parameter)

min: minimum number that the user can enter

max : maximum number that the user can enter

Return value: an integer (row or column number) in the range min…max

Method description

```
• loop until you read a valid number from the keyboard
    o if the number being read from the keyboard is a valid integer number (use hasNextInt)
        ▪ Read it from the keyboard using nextInt method and store it in the variable num
        ▪ if num is between max and min inclusive
              valid is true
          else print error message
      else it is a not a valid integer number
        ▪ Print an error message
```

DELIVERABLES:

- Your program should produce a report similar to that shown below.
- You must create enough empty lines between the runs of the code so that the player would not be able to see the previous board.

Sample output:

```
CSC 15 -   Your Name


Welcome to the Memory Game.
The goal is to find all the matching pairs in as few turns as possible.
At each turn select two different positions on the board to see if they match.

 Press any key to start the game.


    1 2 3 4
   =========
1 | * * * * |
2 | * * * * |
3 | * * * * |
4 | * * * * |
   =========
Where is the first card you wish to see?
           Row: 1
           Col: 1
Where is the second card you wish to see?
           Row: 1
           Col: 3



    1 2 3 4
   =========
1 | 8 * 8 * |
2 | * * * * |
3 | * * * * |
4 | * * * * |
   =========
You found a match


Press any key to continue.
    1 2 3 4
   =========
1 | 8 * 8 * |
2 | * * * * |
3 | * * * * |
4 | * * * * |
   =========
Where is the first card you wish to see?
           Row: 1
           Col: 2
Where is the second card you wish to see?
           Row: 4
           Col: 4



    1 2 3 4
   =========
1 | 8 2 8 * |
2 | * * * * |
3 | * * * * |
4 | * * * 2 |
   =========
You found a match


Press any key to continue.




    1 2 3 4
   =========
1 | 8 2 8 * |
2 | * * * * |
3 | * * * * |
4 | * * * 2 |
   =========
Where is the first card you wish to see?
           Row: 1
           Col: 4
Where is the second card you wish to see?
```

```
           Row: 3
           Col: 3





    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | * * * * |
3 | * * 7 * |
4 | * * * 2 |
   =========
You found a match


Press any key to continue.





    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | * * * * |
3 | * * 7 * |
4 | * * * 2 |
   =========
Where is the first card you wish to see?
           Row: 2
           Col: 1
Where is the second card you wish to see?
           Row: 4
           Col: 2








    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 * * * |
3 | * * 7 * |
4 | * 4 * 2 |
   =========
You found a match


Press any key to continue.






    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 * * * |
3 | * * 7 * |
4 | * 4 * 2 |
   =========
Where is the first card you wish to see?
           Row: 2
           Col: 2
Where is the second card you wish to see?
           Row: 3
           Col: 1







   1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 6 * * |
3 | 6 * 7 * |
4 | * 4 * 2 |
   =========
You found a match


Press any key to continue.











    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 6 * * |
3 | 6 * 7 * |
4 | * 4 * 2 |
   =========
Where is the first card you wish to see?
           Row: 2
           Col: 3
Where is the second card you wish to see?
           Row: 4
           Col: 3






    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 6 3 * |
3 | 6 * 7 * |
4 | * 4 3 2 |
   =========
You found a match


Press any key to continue.
```

```
    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 6 3 * |
3 | 6 * 7 * |
4 | * 4 3 2 |
   =========
Where is the first card you wish to see?
           Row: 2
           Col: 4
Where is the second card you wish to see?
           Row: 4
           Col: 1
```

```
    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 6 3 1 |
3 | 6 * 7 * |
4 | 1 4 3 2 |
   =========
You found a match


Press any key to continue.
```

```
    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 6 3 1 |
3 | 6 * 7 * |
4 | 1 4 3 2 |
   =========
Where is the first card you wish to see?
           Row: 3
           Col: 2
Where is the second card you wish to see?
           Row: 3
           Col: 4
```

```
    1 2 3 4
   =========
1 | 8 2 8 7 |
2 | 4 6 3 1 |
3 | 6 5 7 5 |
4 | 1 4 3 2 |
   =========
You found a match
```

```
Press any key to continue.

        CONGRADULATIONS! You found all the matching pairs!

You did it in 8 turns.
That is your best score so far!


Do you want to start another game? yes

Press any key to start the game.





    1 2 3 4
   =========
1 | * * * * |
2 | * * * * |
3 | * * * * |
4 | * * * * |
   =========
Where is the first card you wish to see?
           Row: 2e
ERROR: 2e is not a valid integer (1...4)
           Row: 3
           Col: 5
ERROR: 5 is not in the valid range (1...4)
           Col: 3





    1 2 3 4
   =========
1 | * * * * |
2 | * * * * |
3 | * * * * |
4 | * * * * |
   =========
First card: (row 3, col 3)
Where is the second card you wish to see?
        Row: 2
        Col: 4



    1 2 3 4
   =========
1 | * * * * |
2 | * * * 6 |
3 | * * 3 * |
4 | * * * * |
   =========
Sorry, No match.
Press any key to continue.




    1 2 3 4
   =========
1 | * * * * |
2 | * * * * |
3 | * * * * |
4 | * * * * |
   =========
Where is the first card you wish to see?
           Row: 5
ERROR: 5 is not in the valid range (1...4)
           Row: 1
           Col: 1




    1 2 3 4
   =========
1 | * * * * |
2 | * * * * |
3 | * * * * |
4 | * * * * |
   =========
First card: (row 1, col 1)
Where is the second card you wish to see?
        Row: 2
        Col: 3




    1 2 3 4
   =========
1 | 4 * * * |
2 | * * 8 * |
3 | * * * * |
4 | * * * * |
   =========
Sorry, No match.
Press any key to continue.
```