ITI 1121. Introduction to Computing II
Winter 2015

Assignment 2 [PDF]
(Last modified on September 6, 2015)

**Deadline: Thursday February 27, 2014, 18:00 (extended deadline)**

Learning objectives

Writing graphical user interfaces in Java involves inheritance and interface. Accordingly, the learning objectives for this assignment are:

- Writing a software application using inheritance, interfaces, and object-oriented programming concepts;
- Designing an application utilizing event-driven programming.

Game

For this assignment, you must implement the graphical user interface (GUI) for the game 8-Puzzle, the smaller version of the game 15-Puzzle. The game board consists of 3×3 tiles with one tile missing. The user can move an adjacent tile into the empty space. To win the game, the user must restore the order of the tiles by repeatedly moving tiles.
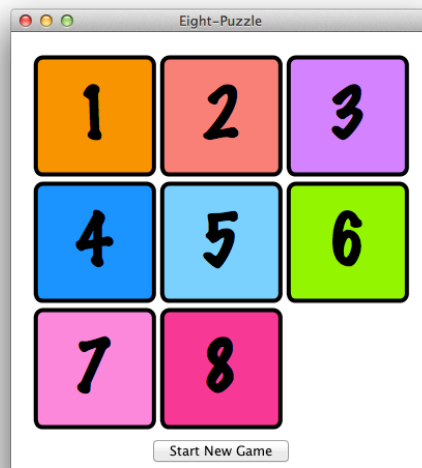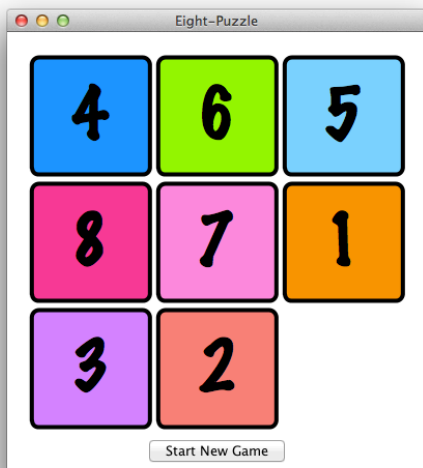
Background information

Instead of starting *de novo* (from scratch), I am asking you to study a similar game, **Puzzler**, for which I am providing you the code and description. Your solution must clearly be an adaptation of the game **Puzzler**

The game **Puzzler** consists of a 7×7 board. Initially, all the cells are filled with marbles from one the five available colors. When the user clicks on a cell for the first time, the cell and all the adjacent cells of the same color are selected. The marbles are gray to indicate this state of the game. When the user clicks a second time on a selected cell, all the selected cells vanish. Marbles fall from top to bottom, and left to right, in order to fill the empty spaces. To win the game, the user must make all the marbles vanish.
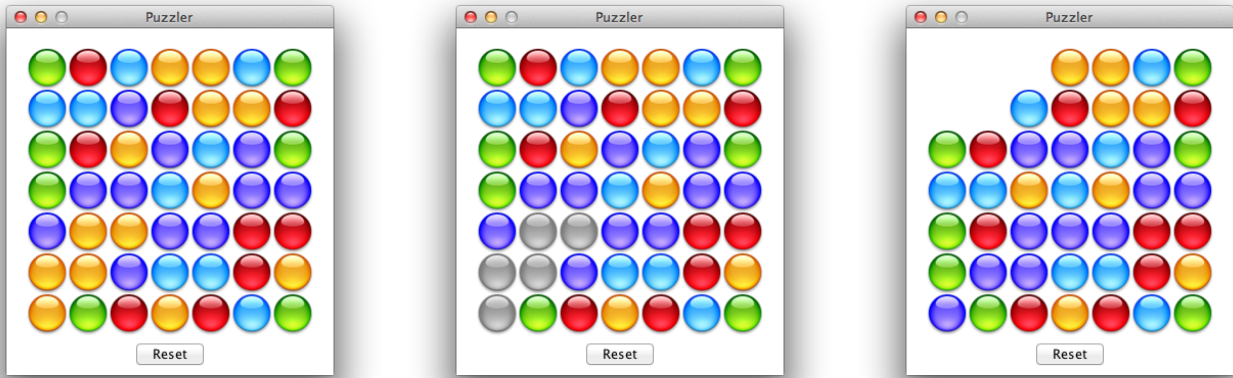
The implementation consists of three classes: Puzzler, Board, and Cell.
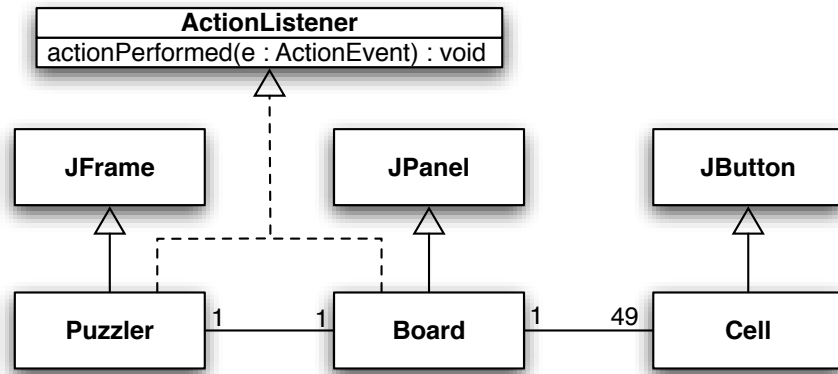
You can clearly see the important role of inheritance and interface for this assignment. Each of the three classes is derived from an existing class. The application needs a main window, which is called here **Puzzler**, and this is a subclass of **JFrame**. We need an object to model the grid onto which all the marbles will be placed. Since the object has access to all the marbles, it will also be responsible for implementing the logic of the game. The class **JPanel** will provide the means to display the marbles, this will be the class **Board**. Finally, the tiles of the **Board** game are implemented using objects of the class **Cell**, a subclass of **JButton**.

The key idea for this implementation is as follows. When a cell vanishes, it simply displays a white square. In order to simulate marbles falling from top to bottom, and left to right, we simply change the type of the source and destination cell. For instance, if a blue marble at position (i, j) moves to position (i', j'). We simply set the type of the cell at position (i, j) to represent the empty cell, whereas the type of the cell at position (i', j') becomes that of a blue marble.

The UML sequence diagram on the page illustrates some of the important sequences of method calls. When the constructor of the class **Puzzler** is called, it will create an object of the class **Board**. The constructor of the class **Board** will itself create **Cell** objects. The diagram shows one such object creation. The constructor of the class **Cell** receives a reference to the **Board** object, which will serve as an action listener for this **Cell** (call to addActionListener(b)).

The second part of the diagram illustrates a possible sequence of method calls resulting from a mouse click. When the user clicks the button, an object is created to represent this action, the method **processEvent** of the button is called. The button will then call the method **actionPerformed** of its action listener (the object that was registered using the method **addActionListener**, as above). The board will determine the source of the event. The method **actionPerformed** interacts with the cell to determine its type, row, and column. If the cell was not selected, all the cells are deselected, and all the adjacent cells are selected, this will include a class to the method **setSelected** of the selected cell, which changes the image to that of a gray marble. Eventually, the control returns to the caller.
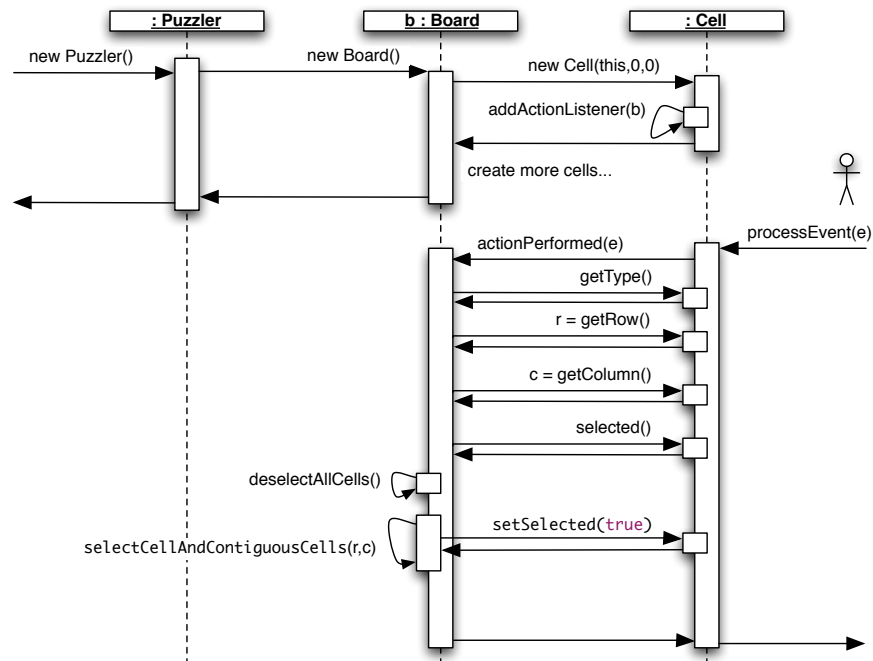
1 Rules and regulation (15 marks)

Follow all the directives available on the assignment directives web page, and submit your assignment through the on-line submission system uottawa.blackboard.com. You must preferably do the assignment in teams of two, but you can also do the assignment individually. Pay attention to the directives and answer all the following questions.

2 EightPuzzle (20 marks)

An object of the class **EightPuzzle** represents the main window of the game. It is displayed from the beginning to the end of the game. The top part of the window displays the grid holding the board, whilst the lower part comprises a button, labeled "Start a new game", which is used to reinitialize the game and put tiles in random order. Here are the characteristics of the class.

- It is a subclass of the class **JFrame**.

- It implements the interface **ActionListener**. Consequently, it must implement the method **actionPerformed(ActionEvent e)**. Make sure that whenever the user clicks on the "Start a new game" button, this method is called.

- An object of the class **EightPuzzle** has a reference to an object of the class **Board**.

- Finally, the class has a **main** method that starts the execution of the game.

3 Board (35 marks)

Board is a specialized **JPanel** that represents the grid of 3×3 cells (objects of the class **Cell**). The class **Board** implements the game logic. An object of the class **Board** is the handler for the events generated by the objects of the class **Cell**. Here are the characteristics of the class:

- It is a sub-class of the class **JPanel**.
- The class **Board** implements the interface **ActionListener**.
- A **Board** object must create all the necessary **Cell** objects to fill the grid. It must save references to these **Cell** objects.
- A **Board** object must keep track of the total number of attempts since the last reset of the game.

Here are the mandatory methods.

- A method **init** that resets the game and places all the tiles in random order, with the empty cell in the lower right corner.

- **actionPerformed(ActionEvent e)**, as required per the contract of the interface **ActionListener**. An object of the class **Board** is the action listener for all the events generated by the Cells.
- **String toString()** returns a String representation of the board. Each cell shows the id of the cell.

```
[2][8][7]
[5][3][4]
[6][1][0]
Number of moves is 0
```

Other requirements

- When the user has solved the puzzle, i.e. the tiles are in order, and the empty cell is on the lower right side of the board, display a message to user indicating the total number of attempt, and reset the game. Suggestion, use **JOption-Pane.showMessageDialog** to display a dialog.

4 RandomPermutation (10 marks)

An instance of the class **RandomPermutation** is used to create a randomly generated permutation. The permutation consists of the numbers 0 to (row × columns − 1) in random order, with the property that the 0 must always be the last element of the permutation. In this application, 0 is the type of the empty cell, at the start of a new game the empty cell is always at the lower right corner.

We would like the newly created permutation to be the identity permutation, with the zero in last position. This might be useful for debugging your application. Executing the following statements:

```
RandomPermutation p;
p = new RandomPermutation(3,3);
System.out.println(p);
p.shuffle();
System.out.println(p);
```

produces the following output.

```
[1][2][3]
[4][5][6]
[7][8][0]

[2][1][3]
[5][6][4]
[8][7][0]
```

The are two allowed implementations, worth a maximum of 5 and 10 marks, respectively. The first implementation generates solutions that might be unsolvable, whereas the second one always generates solvable solutions. Suggestion: implement the first solution, then come back to this question and consider the second implementation if time allows.

Implementation 1 (maximum 5 marks)

For this implementation, a call to the method shuffle simply generates a randomly generated permutation of the numbers 1 to (row × columns − 1), followed by zero. This permutation may or may not be solvable.

- Has a constructor **RandomPermutation(int row, int column)**, where **row** and **column** specify the size of the board.

- **int[] toArray():** returns this permutation in an array, where the elements of the first row comes first, followed by the elements of the second row, etc.

- **shuffle():** simply generates a new permutation. The permutation is not guaranteed to be solvable.

- **String toString():** returns a String representation of the permutation (see above for examples).

Implementation 2 (maximum 10 marks)

For this implementation, a call to the method shuffle must guarantee that the solution is solvable. Johnson an colleagues proposed a mathematical framework for generating such permutations. Herein, we explore a computational approach.

- Has a constructor **RandomPermutation(int row, int column)**, where **row** and **column** specify the size of the board.

- **int[] toArray():** returns this permutation in an array, where the elements of the first row comes first, followed by the elements of the second row, etc.

- **shuffle()**: returns a permutation that is guaranteed to be solvable. The approach is simple. Starting from the identity permutation, generate a large number of randomly generated moves. Imagine that instead of moving a tile, you are moving the empty space. In my implementation, I am applying 1000 random moves.

- **String toString():** returns a String representation of the permutation (see above for examples).

References

- Johnson, Wm. Woolsey; Story, William E. (1879), "Notes on the 15-Puzzle", American Journal of Mathematics. **2** (4): 397404.

5   Cell (20 marks)

The class **Cell** is derived from the class **JButton**. An object of this class represents a **Cell** on the grid. A **Cell** has an **id**. This **id** represents the image that should be used to represent the button. A button memorizes its coordinates on the board.

- A constructor having three parameters, one of type **Board** and the others to represent the row and column of this object. The object of the class **Board** will be the event listener of this specialized **JButton**.
- **getId()**: returns the id of the cell.
- **setId(int id)**: sets the id of the cell and updates the image accordingly.
- **getRow()**: returns the value of the attribute row.
- **getColumn()**: returns the value of the attribute column.
- **String toString()**: returns the id of the cell, represented as a String.

Bonus (10 marks)

- Make the game more general by allowing the user to select the number of rows and columns. (5 marks)

- Load an image from a file, break the image in $m \times n$ tiles, which you will then use in place of the tiles provided with the assignment. (5 marks)

Advanced topic

Consider adding a method that finds the minimum number of moves for solving the game.

Files

The directory **data** contains images representing the tiles for the 8-Puzzle game: data (data.jar). You must hand in a working application, make sure to include the image directory, and all the other required files. You must hand in the following files:

- README.txt
- StudentInfo.java
- EightPuzzle.java
- Board.java
- Cell.java
- RandomPermutation.java
- data

Here are two versions of an application called **Puzzler**. Do not hand in Puzzler when submitting your solution.

- puzzler-src.jar (**Basic**)

- Puzzler.jar (**Advanced**)

These two archives contain the source code, the byte-code, as well as the images. The first implementation does not declare a package. In order to execute the application, you must execute the main method of the class **Puzzler** from the directory where the sub-directory **data** is found. The second implementation declares a package and contains a manifest file. The file manifest.mf specifies the name of the class that contains the method method. Because of that, you can double-click the .jar file to start the application.

WARNING: Puzzler.jar contains some advanced features to make it executable. Consider using **puzzler-src.jar** first.

A   Frequently Asked Questions (FAQ)

1. **"Are we allowed to introduce new methods in existing classes and possibly introducing new classes?"**

   - **If you are not answering the bonus question,** then you cannot introduce new public methods. **Board**, and **Cell**.
   - **If you are answering the bonus question,** then you can introduce new public methods.

2. **"The tiles are not showing, can you help?"**

   What did you do to locate the source of the problem? For instance, did you write a test just for cell.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class CellTest extends JFrame implements ActionListener {

    public CellTest() {
        super("Cell Test");
        Cell c;
        c = new Cell(null, 0, 0);
        c.setId(4); // how we control the color
        add(c,BorderLayout.CENTER);
        JButton button;
        button = new JButton("Zzz");
        button.addActionListener(this); // registering this CellTest object with the butt
        add(button,BorderLayout.SOUTH);
        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        System.out.print("Howdy!");
    }

    public static void main(String[] args) {
        new CellTest();
    }
}
```

Did you notice that the file names for the images contain an extra 0?

Last Modified: September 6, 2015