PGCert IT: Programming for Industry

Practical Quiz 04

Time allowed: **two hours** (plus 10 minutes reading time)

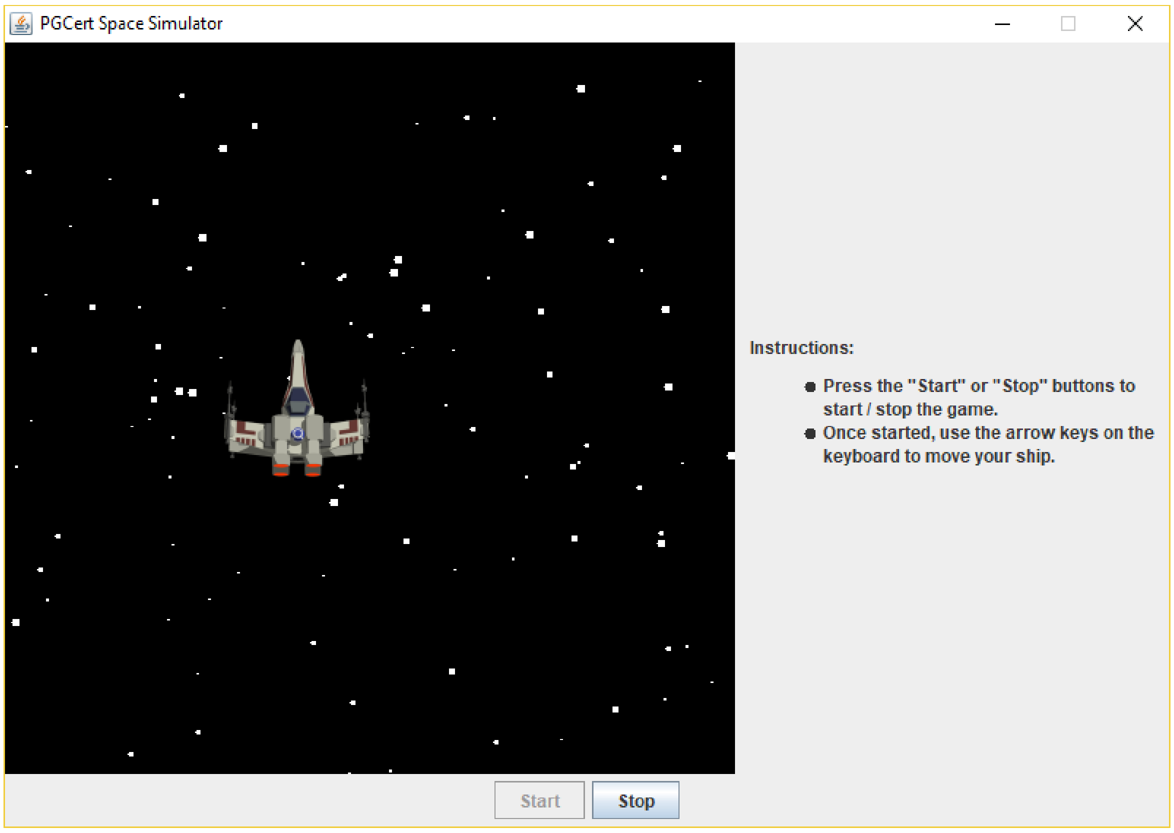
Notes:

- There are **two questions** in this practical test, worth 30 **marks each**. Please attempt **both questions**.
- This is an **open book** test. You may use any bound printed or handwritten notes during the test. You may also use any online or PC-based resources available to you - but **no use of email, chat programs, or attempting to solicit online assistance is permitted**.
- You will be provided with a Zip file containing an IntelliJ project which will serve as a starting point for the test. The project will contain **two sub-projects** - one for each question - to prevent any compile errors from one question affecting your ability to test the other!
- To submit your answers, Zip your project and submit the single Zip file to Canvas or Moodle at or before the end of the test.
- **Important:** Make sure to read all instructions carefully before attempting each question.

Question One - A Space Game

In this question, we'll complete the code for a simple game which allows the user to fly a 2D spaceship around the screen. The completed app is shown in the screenshot below.

When the user clicks the "start" button, the starfield in the background will animate. Then, the user will be able to fly the ship up, down, left, or right by pressing the corresponding arrow key on the keyboard. The ship's graphic will change to indicate the direction it is currently flying. The user will not be able to move the ship outside the visible area - if they try, the ship will stop at the edge.



Instructions:

- Press the "Start" or "Stop" buttons to start / stop the game.
- Once started, use the arrow keys on the keyboard to move your ship.

The app, as given to you, will simply render a black square - you won't be able to see the stars or ship. To complete the app, perform the following tasks:

1. **Complete the `Star` class.** This class is intended to represent a single star in the starfield. It contains variables representing its position, size, and speed. To complete this class:

    a. Implement the `move()` method. Each time this method is called, the star should move *down* by a number of pixels specified by its *speed*. Then, if the star has moved off the bottom of the screen (whose height is given by the `windowHeight` variable), it should appear back at the top of the screen.

    b. Implement the `paint()` method. Each time this method is called, a *white filled oval* should be drawn at the star's location. The width and height of the oval should both be the star's size.

2. **Complete the `Starfield` class.** This class is intended to represent all the stars in the game. It contains a `List` of `Star` objects to be animated / drawn. To complete this class:

   a. Implement the constructor. This constructor should appropriately initialize the `stars` list, and fill it with 100 stars. Each star should have:
      i. A random x, between 0 (incl.) and the given width (excl.)
      ii. A random y, between 0 (incl.) and the given height (excl.)
      iii. A random size, between 2 (incl.) and 6 (incl.)
      iv. A random speed, between 5 (incl.) and 11 (incl.)

   b. Implement the `move()` method. This method should loop through all the stars and appropriately call their `move` methods.

   c. Implement the `paint()` method. This method should loop through all the stars and appropriately call their `paint` methods.

3. **Complete the `paintComponent()` method in the SpacePanel class.** Currently, this method simply draws a black rectangle. Modify the method so that it additionally paints the starfield and the spaceship. The spaceship should be drawn on top of the starfield. Once you've done this, you should be able to run the app and see these objects on-screen (though they won't move yet).

4. **Add a `Timer`.** Currently, when the "start" button is clicked, `SpacePanel`'s `start()` method is called (the code for this is in the `SpaceApp` class). Similarly, `SpacePanel`'s `stop()` method is called when the "stop" button is clicked. But these methods don't do anything yet. We'll add a Swing `Timer` to `SpacePanel` to allow the game to animate. To do this:

   a. Have `SpacePanel` implement the `ActionListener` interface. Then, in the appropriate method, call the starfield's `move()` method. Then, call the appropriate method which will allow the game to *repaint* itself. **Hint:** The width and height of `SpacePanel` can be obtained using its `getWidth()` and `getHeight()` methods.

   b. In `SpacePanel`'s constructor, initialize the `timer` instance variable to a new timer, with a delay of 20ms, and "this" as its action listener.

   c. In `SpacePanel`'s `start()` method, start the timer. In its `stop()` method, stop the timer. Once this is done, you should be able to run the app and click the "start" button to see the starfield animate. The animation should stop when the "stop" button is clicked.

5. **Move the `Spaceship` (part 1).** Currently, there's a variable in `SpacePanel` called `moveDirection`, which is intended to store the direction the spaceship should move each time the timer ticks. We want to change this value based on the user's key presses, and supply it to the `Spaceship` class, which should then move appropriately. To do this:

a. Have `SpacePanel` implement the `KeyListener` interface. In an appropriate method, set the value of the `moveDirection` variable based on the *key code* of the supplied `KeyEvent`. If the key code is `VK_UP` (the "Up" arrow on the keyboard), `moveDirection` should be set to `Direction.Up`. Similar key codes for the other three keyboard arrows should be mapped to the other three directions. If any other key code is detected, `moveDirection` should be set to `Direction.None`. `moveDirection` should also be set to this value when the user *releases* a key.

b. At an appropriate location within the method you added in step 4, add code which will call Spaceship's move method, supplying appropriate arguments.

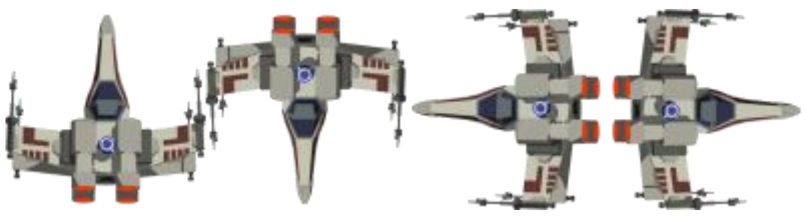
c. Also within this method, add a call to the `requestFocusInWindow()` method. This will allow the panel to continue to receive key press events even if the user clicks another component (for example, the buttons).

d. At an appropriate location within the code, remember to add `SpacePanel` as a *key listener* on itself.

6. **Move the `Spaceship` (part 2).** Now that `Spaceship`'s `move()` method is being called appropriately, we need to implement it. If the supplied direction is `None`, the ship shouldn't move. Otherwise, it should move in the specified direction a number of pixels equal to its `speed`. If the ship would go out-of-bounds (based on the provided `windowWidth` and `windowHeight`), it should stop just at the edge. If the ship did move, you should also store the direction it moved in the `direction` instance variable - this will be useful later.

7. **Change ship graphic based on direction.** Currently, the ship's `paint()` method will always draw the spaceship facing upwards. However, spaceship.png contains the graphics for all four directions and we would like to use them. Modify the `paint()` method to take the ship's `direction` into account.

   Hint: Refer to the JavaDoc for more information about the `drawImage()` method used here, which is capable of drawing *a portion* of a source image onto a destination location.



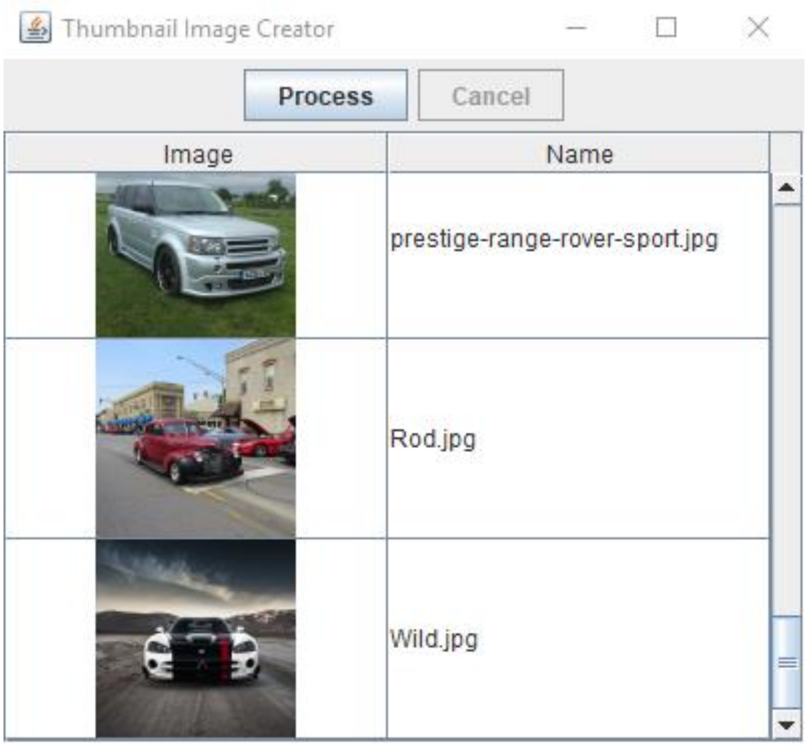
Marking Guide

Question one is marked according to the following criteria:

Star class	*(3 marks)*

Starfield class	*(2 marks)*
paintComponent	*(2 marks)*
Implements ActionListener	*(3 marks)*
Create, start and stop a Timer	*(3 marks)*
Correctly implement KeyListener	*(4 marks)*
Movement logic (in SpacePanel)	*(4 marks)*
Movement logic (in Spaceship)	*(4 marks)*
Ship draws differently based on direction	*(5 marks)*

Question Two - A Thumbnail Generator



Class `ThumbnailGeneratorApp` is a Swing application that generates thumbnails from full size images. In response to pressing the Process button, the application allows the user to select a directory that should contain images with a .jpg extension. After selecting the directory, the application generates a thumbnail for each image and stores them in a subdirectory named thumbnails. The app should also populate a table showing each thumbnail it has generated.

Currently, the program correctly performs the thumbnail generation – you can go ahead and run it from within IntelliJ to see what it does, and you may use the supplied `pictures` folder containing car images for testing purposes. However, for large image files in particular, the processing can be expensive and `ThumbnailGeneratorApp` handles all image processing on the Event Dispatch thread. This means the UI is completely unresponsive while processing - the user's only feedback is a "success" message displayed at the end. Furthermore, while the generated thumbnails are added to a `ThumbnailList` object, the contents of this list is not displayed in the `JTable`.

Your tasks for this exercise are to modify the program so that the image processing is done in a background thread, and the `JTable` is appropriately updated.

Begin by thoroughly reading through and understanding the code - particularly that within the `ThumbnailGeneratorApp` class and classes within the `model` package - then proceed with parts A and B of this question.

Part A - JTable Display

By examining the code, you'll see that each time a Thumbnail is created, it will be added to a `ThumbnailList` object. This object follows the *observer* pattern, and is capable of notifying its observers - instances of `ThumbnailListener` - whenever items are added or removed from it.

You'll also notice that the JTable's model is set to an instance of `ThumbnailTableModelAdapter`, and provided with the `ThumbnailList`. However, the adapter class is largely unimplemented. For part A, you'll complete this class by performing the following tasks:

1. Allow the `JTable` to display the information in a `ThumbnailList` by completing the implementation of the `getValueAt()`, `getRowCount()`, and `getColumnCount()` methods in the adapter class. You'll also need to store a reference to the `ThumbnailList` object that's provided in the constructor.

2. Allow the `ThumbnailList` to notify the adapter class whenever it changes. This can be done by having the adapter implement an appropriate interface, and by having it register itself as a listener on the `ThumbnailList`.

3. Whenever the `ThumbnailList` does change, the adapter needs to notify the JTable that its data has changed. To do this, make appropriate use of the `fire…()` methods.

Part B - SwingWorker

Once part A is complete, the generated thumbnails will be displayed in the table. However, the app is still unresponsive, and the thumbnails are only displayed once all processing is complete. Your task is to modify the program so that it more appropriately uses a `SwingWorker` to handle image processing.

Specifically, modify `ThumbnailGeneratorApp` such that it meets the following requirements:

1. All image loading, thumbnail generation and saving must be performed in the background. Code in lines 102 to 116 should execute on a background thread, while code in lines 118 to 127 should execute on the event dispatch thread once processing is complete. These areas are marked with TODO statements.

2. Whenever an individual thumbnail has been generated, the GUI should report (as an intermediate result) that that thumbnail has been generated. Your background thread should `publish()` individual thumbnails to the ED thread, where they can be `process()`'d.

3. By clicking on the Cancel button, the user must be able to abort the processing. Use SwingWorker's cancellation mechanisms for this purpose. If the operation is cancelled, a dialog box should be displayed to the user informing them of this.

Marking Guide

Question two is marked according to the following criteria:

Adapter correctly allows JTable to display thumbnails	*(5 marks)*
Adapter correctly listens to changes in ThumbnailList	*(3 marks)*
Adapter correctly fires table-changed events to notify the JTable it should update.	*(1 marks)*
Nested SwingWorker class definition	*(3 marks)*
Appropriately starts SW running	*(3 marks)*
Necessary code moved into various SW methods	*(5 marks)*
No incorrect code moved into SW methods	*(2 marks)*
Intermediate results handled correctly	*(3 marks)*
SW can be cancelled	*(4 marks)*
GUI “niceties” (enabling / disabling buttons, displaying alternative mouse cursor, etc)	*(1 marks)*